AFRL-RI-RS-TR-2015-171

FORMAL VERIFICATION OF QUASI-SYNCHRONOUS SYSTEMS

ROCKWELL COLLINS

JULY 2015

FINAL TECHNICAL REPORT

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

AFRL-RI-RS-TR-2015-171 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/ S /	**/ S /**
STEVEN L. DRAGER	MARK H. LINDERMAN
Work Unit Manager	Technical Advisor, Computing
	& Communications Division
	Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)* JULY 2015	**2. REPORT TYPE** FINAL TECHNICAL REPORT	**3. DATES COVERED** *(From - To)* JAN 2013 – JAN 2015

4. TITLE AND SUBTITLE FORMAL VERIFICATION OF QUASI-SYNCHRONOUS SYSTEMS	**5a. CONTRACT NUMBER** FA8750-13-C-0051
	5b. GRANT NUMBER N/A
	5c. PROGRAM ELEMENT NUMBER 63781D
6. AUTHOR(S) Steven P. Miller, Sidhartha Bhattacharyya, Cesare Tinelli, Scott Smolka, Christoph Sticksel, Baoluo Meng, Junxing Yang	**5d. PROJECT NUMBER** ASET
	5e. TASK NUMBER 12
	5f. WORK UNIT NUMBER RC
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Rockwell Collins, 400 Collins Road NE, Cedar Rapids, IA 52498	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/RITA, 525 Brooks Road, Rome NY 13441-4505	**10. SPONSOR/MONITOR'S ACRONYM(S)** AFRL/RI
	11. SPONSOR/MONITOR'S REPORT NUMBER AFRL-RI-RS-TR-2015-171

12. DISTRIBUTION AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited. PA# 88ABW-2015-3369
Date Cleared: 2 Jul 15

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Modern defense systems must be implemented as redundant, fault-tolerant systems in order to meet their reliability requirements. Unfortunately, developing protocols to achieve distributed agreement in an asynchronous environment can be deceptively difficult. Engineers often exploit the fact that their systems are quasi-synchronous, where even though the clocks of the different nodes are not synchronized, they run at the same rate, or multiples of the same rate, modulo their drift and jitter. While such designs often appear to work correctly, their intrinsic asynchrony makes them prone to latent race and deadlock conditions. This project provide systems designers with an intuitive modeling environment that 1) allows systems engineers to easily specify the high-level architecture and synchronization logic of quasi-synchronous systems using widely available system engineering notations and tools, and 2) integrates and enhances innovative formal verification tools to provide them with immediate feedback on the correctness of their designs

15. SUBJECT TERMS

Distributed systems, distributed agreement, quasi-synchronous, formal methods, architecture, model checking, SysML, AADL, AGREE, Kind, UPPAAL

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			STEVEN L. DRAGER
U	U	U	UU	105	**19b. TELEPHONE NUMBER** *(Include area code)* N/A

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Table **Page**

1.0 SUMMARY

Modern defense systems are complex software systems implemented over heterogeneous and constantly evolving hardware and software platforms. Due to the failure rates of individual hardware components, critical functions must be implemented as redundant, fault-tolerant systems in order to meet their reliability requirements. This is achieved by distributing these functions over multiple processing components connected by fault-tolerant networks. When a system is replicated to achieve a high level of reliability, the individual components still need to agree on some part of the global system state, such as which node is the current leader. While the amount of state that needs to be consistent is often tiny, that consistency is essential for the correct behavior of the system. Unfortunately, developing protocols to achieve agreement in an asynchronous environment can be deceptively difficult.

In developing distributed agreement protocols, engineers often exploit the fact that their systems are *quasi-synchronous*, where even though the clocks of the different nodes are not synchronized, they all run at the same rate, or multiples of the same rate, modulo their drift and jitter. While such designs often appear to work correctly, their intrinsic asynchrony makes them prone to latent race and deadlock conditions. These design errors often do not appear until late in system integration or even after the system is deployed.

The goal of this project was to provide systems designers with an intuitive modeling environment that 1) allows systems engineers to easily specify the high-level architecture and synchronization logic of quasi-synchronous systems using widely available system engineering notations and tools, and 2) integrates and enhances innovative formal verification tools such as Satisfiability Modulo Theories (SMT) based model checkers and model checkers for timed automata to provide system engineers with immediate feedback on the correctness of their designs.

In the modeling environment, system developers create high-level models of the system architecture and synchronization logic using the Enterprise Architect System Modeling Language (SysML) modeling environment enhanced with a SysML profile for quasi-synchronous systems. A translator translates these models into the Architectural Analysis and Description Language (AADL) and imports them into the Open Source AADL Tool Environment (OSATE). System properties can then be verified using either the Assume Guarantee Reasoning Environment (AGREE) with the Kind model checker or the Uppaal model checker for timed automata.

Four examples of quasi-synchronous systems were created and verified: the Pilot Flying example, the Leader Selection example, the Active Standby example, and the Wheel Breaking System (WBS) example. All of these are based on actual examples seen in industry. The WBS example is derived from an accident report in which a commercial air transport aircraft lost all braking capability on landing. Critical synchronization constraints for these examples were formally specified and verified using both Kind and Uppaal. In the case of the WBS example, the logic synchronization error that caused the loss of braking was readily found. In addition, another counterexample not discussed in the accident report was found using the Kind model checker.

These examples clearly demonstrate the difficulty of developing correct distributed agreement protocols without the use of formal verification tools that examine all possible combinations of inputs and states. They also show that distributed agreement protocols can be simplified by exploiting the quasi-synchronous relationship of the clocks found in many real systems.

2.0 INTRODUCTION

Modern defense systems are complex software systems implemented over heterogeneous and constantly evolving hardware and software platforms. Due to the failure rates of individual hardware components, critical functions must be implemented as redundant, fault-tolerant systems in order to meet their reliability requirements. This is achieved by distributing these functions over multiple processing components connected by fault tolerant networks. When a system is replicated to achieve a high level of reliability, the individual components still need to agree on some part of the global system state, such as which node is the current leader. While the amount of state that needs to be consistent is often tiny, that consistency is essential for the correct behavior of the system. Unfortunately, developing protocols to achieve agreement in an asynchronous environment can be deceptively difficult. In fact, the difficulty of doing this is so well known that entire textbooks have been written on the design of distributed algorithms [1], [2].

Too often, this activity is treated as a low-level software design problem rather than as the high-level systems engineering task it actually is. Moreover, when designing the synchronization logic, engineers often exploit the fact that their systems are *quasi-synchronous* systems, where even though the clocks of the different nodes are not synchronized, they all run at the same rate, or multiples of the same rate, modulo their drift and jitter [3], [4]. Such designs often appear to work correctly, but their intrinsic asynchrony makes them prone to latent race and deadlock conditions. These design errors often do not appear until late in system integration or even after the system is deployed.

Commercial modeling tools are starting to emerge for the design of complex system architectures, but this problem can actually be made worse if these tools are not used with care. System designers will often model their distributed system architecture and synchronization logic using popular commercial tools such as MATLAB and Simulink/Stateflow® [5] [5] or the Safety Critical Application Development Environment (SCADE) Suite™ of Esterel Technologies [6], failing to realize that these tools assume an underlying synchronous model of computation in which all nodes are driven by a single global clock. Even though their designs appear to work correctly when simulated in the modeling environment, the actual implementations are likely to fail since they are implemented on processors executing asynchronously.

This project provides systems designers with an intuitive modeling environment that 1) allows systems engineers to easily specify the high-level architecture and synchronization logic of quasi-synchronous systems using widely available system engineering notations and tools, and 2) integrates and enhances innovative formal verification tools such as SMT-based model checkers and model checkers for timed automata to provide system engineers with immediate feedback on the correctness of their designs. An important aspect of this approach is that once the high-level system architecture and synchronization logic are verified, that system model can be refined to become the complete system design without invalidating the correctness of the initial design. An overview of this environment is shown in Figure 1.

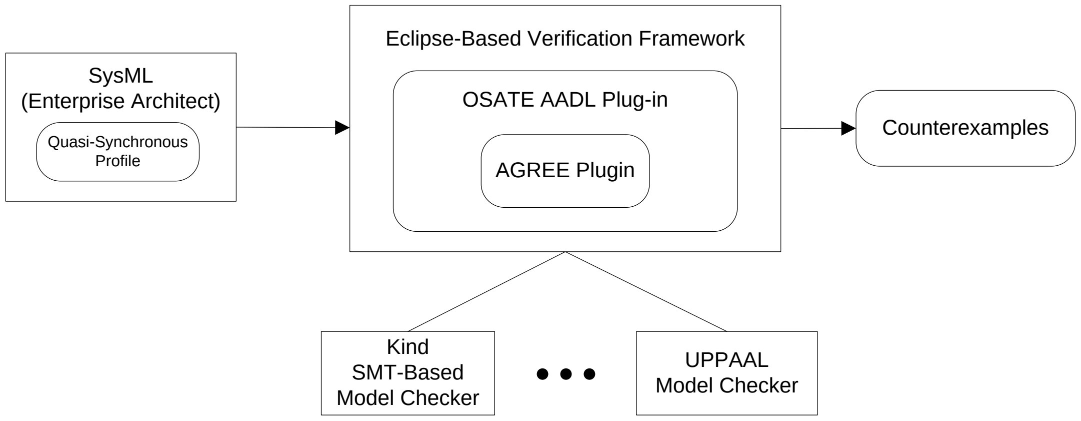


Figure 1 – Framework for the Verification of Quasi-Synchronous Systems

To provide a realistic path for technology transfer, system developers can create high-level models of the system architecture and synchronization logic using the Enterprise Architect SysML [7], [8] modeling environment enhanced with a SysML profile for quasi-synchronous systems. Translators import these models into the OSATE development environment for AADL [9]. Component behaviors specified as SysML state machines are translated into AGREE and Behavior Annex (BA) specifications within AADL. Verification of the AADL model supplemented with the AGREE annexes can be performed directly using the AGREE tool [10], [11], where AGREE can be configured to invoke either the Kind [12] or jKind[1] SMT-based model checker. Verification of the AADL model supplemented with the Behavior Annexes can be performed using the Uppaal model checker [13] for timed automata by first invoking a translator that converts the AADL and Behavior Annex specifications into an Uppaal model. This model can then be verified using the graphical user interface provided with the Uppaal tool.

The remainder of this report is structured as follows. Section 3.1 provides background information on SysML, AADL, Kind, and Uppaal. Section 3.2 describes the notion of quasi-synchrony and its relevance to the problem at hand. Section 3.3 introduces the four example problems. Section 3.4 discusses the translations from SysML to AADL and from AADL to Kind and Uppaal. Section 4.1 discusses the verification of the four examples with Kind and Section 4.2 discusses the verification of the four examples with Uppaal. Finally, Section 5.0 summarizes the project, presents our conclusions, and identifies directions for future research.

[1] jKind is light-weight version of Kind provided with the AGREE tool.

3.0 METHODS, ASSUMPTIONS, AND PROCEDURES

This chapter describes the methods, assumptions, and procedures followed in conducting this work. Section 3.1 provides background material on the modeling languages and analysis tools on which the project is based. Section 3.2 explains in detail the concept of quasi-synchrony and its relevance to the verification of distributed systems. Section 3.3 describes the four example problems developed to demonstrate and exercise the tool framework. Finally, Section 3.4 describes how models are automatically translated from the original architectural models to the formal verification tools.

3.1 Background Material

This section provides background material on the modeling languages and analysis tools on which the project is based. Section 3.1.1 describes the relevant aspects of the SysML architectural modeling language used by the system designers to specify a system and its synchronization protocols. Section 3.1.2 describes AADL and its use as an intermediate language in the conversion from SysML to the formal verification tools. Section 3.1.3 describes the Kind SMT-based model checker and Section 3.1.4 describes the Uppaal model checker for timed automata, both of which are used for formal verification of the quasi-synchronous systems.

3.1.1 System Architectural Modeling with SysML.

Notations and tools for the modeling of system architectures are just starting to emerge, with the most widely used today being SysML. SysML is an Object Management Group (OMG) standard for the specification, analysis and design of a broad range of complex systems and systems of systems [7]. It is defined as an extension of a subset of the Unified Modeling Language (UML). Like UML, SysML is a graphical modeling notation, not a methodology or a tool. It makes use of seven of UML's thirteen diagram types (the other six diagram types were felt to be too software-specific and were omitted from SysML). Of the seven included diagram types, three are devoted to modeling the system structure. These include:

Package Diagram: Package Diagrams are used to organize a model and group model elements into a name space. Packages often appear in the navigation windows of tool browsers. Packages can be imported to reduce the need to use fully qualified names in a model.

Block Definition Diagram: Blocks are a fundamental construct used to describe the structure of an element or system. To reuse complex relationships among blocks, a Block Definition Diagram can be defined once to specify a pattern of blocks so that the pattern can then be reused. A block can be defined with associated properties, operations, constraints, allocations, and requirements.

Internal Block Diagram: The internal structure of a block is described using an Internal Block Diagram. This kind of diagram defines the Parts and the interconnection of Parts using Ports, Connectors, and Flows. The Internal Block Diagram most closely matches the traditional notion of an architecture.

Four of the seven diagram types included from UML are devoted to modeling the behavior of the system, either of components of the system or the interactions between components. These include:

Activity Diagram: Activity Diagrams are used to specify the sequence and control of activities that transform inputs to outputs, somewhat analogous to flow charts, but with constructs that allow activities to occur in parallel, similar to Petri Nets. Swim lanes can be used to show which objects perform which activities.

Sequence Diagram: Sequence Diagrams are used to specify how parts of a block interact by exchanging messages. Block elements are aligned across the top of the diagram with a vertical lifeline running downwards indicating increasing time. The types and order of messages exchanged are shown as horizontal arrows between the object’s lifelines. Various structures are used to indicate synchronous (blocking) communication, asynchronous (non-blocking) communication, conditional guards, etc.

State Machine Diagram: State Machine Diagrams define the behavior of a block using a Statecharts-like notation. The major states of the block are depicted along with transitions between states. Transitions from one state to another are triggered by events with optional guards and actions that can be performed. Concurrent and sequential execution of state machines is supported.

Use Case Diagram: Use Cases are used to describe the functionality of a system in terms of how its users use the system to achieve their goals. They provide an informal way of describing a dialogue of requests and actions between the actors and the system to achieve some goal. Use cases can be used to describe the “sunny day” dialogue in which nothing goes wrong as well as alternate courses and exception cases.

With respect to UML, SysML includes two new diagram types to meet the specific needs of systems modeling and analysis. The first of these, the Parametric Diagram, is a structural diagram:

Parametric Diagram: Parametric Diagrams are used to capture the constraints on the properties of the system, where the constraints are expressed as equations whose parameters are bound to the properties of the system. These can then be used by analysis tools to support engineering trade-offs.

The second new diagram, the Requirements Diagram, is used to capture the requirements contained in a specification:

Requirements Diagram: Requirement Diagrams represent text-based requirements. They include an id and text and can be used to specify functional, interface, and performance requirements. In addition, relationships can be defined to show that a requirement is derived from, is satisfied by, is verified by, refines, traces to, and copies some other requirement or entity (e.g., a test case).

SysML can be further extended or constrained through the use of stereotypes, tagged values, and constraints. Stereotypes allow a user to extend a meta-class with additional tagged values and constraints. A profile is a collection of such extensions that collectively customize UML or SysML for a particular domain.

SysML was designed as a general-purpose systems modeling language, but it can also be used to model embedded real-time systems. In the Defense Advanced Research Projects Agency (DARPA) META program [10], we demonstrated that a high-level model of the system architecture can be created using only SysML package, block definition, and internal block diagrams. Interactions between architectural components can be specified using SysML ports and connections. These basic SysML components can be further extended with the definition of a profile containing stereotypes for entities found in embedded real-time systems such as threads and processors. The synchronization logic of the system can be specified using the state machine diagrams of SysML. This provides a natural modeling style for system designers, while emphasizing that design of the synchronization logic is a system engineering activity distinct from specifying the behavior of individual components.

3.1.2 System Architectural Modeling with AADL.

The Architectural Analysis and Design Language is an international standard (AS5506) of the Society of Automotive Engineers (SAE) [9], [14]. AADL is used to model the software and hardware architecture of embedded, real-time systems, but is not limited to the automotive domain. In fact, it was originally developed for avionics systems. It is derived from MetaH, an architecture description language developed by the Advanced Technology Center of Honeywell.

Due to its emphasis on the embedded domain, AADL contains constructs for modeling both software and hardware components. AADL contains equivalent textual and graphical standards, and it is possible to automatically convert from one to the other. It is possible to extract many different views of an AADL model, but there is only one underlying model of the system, ensuring consistency of the different views.

The basic modeling constructs, or component categories of AADL are divided into three groups as shown in Table 1. The first group is concerned with modeling the application software and includes constructs for software components. The second group is concerned with modeling the hardware execution platform and includes constructs for modeling hardware components. The third supports the composition of components into subsystems and systems. It consists of a single *System* construct.

Table 1 – AADL Component Categories

Software Constructs	
Thread	Unit of concurrent execution.
Thread Group	Compositional unit for organizing threads.
Process	Protected address space.
Data	Data types and static data in source text.
Subprogram	Callable sequentially executable code.
Hardware Constructs	
Processor	Entity that schedules and executes threads.
Memory	Location for storing code and data.
Device	Sensors, actuators, or other components that interface with the external environment.
Bus	Entity that interconnects processors, memory, and devices.
System Constructs	
System	A compositional unit for integrating other components into distinct units within the architecture.

A system represents a composition of software, hardware, or system components. A system can be organized into a hierarchy of systems to represent a complex system of systems. Within a system, application components can be mapped onto hardware execution components through binding relationships. For example, a thread can be bound to a processor for execution and a process can be bound to memory. Constructs are available to specify what types of bindings are allowed and to ensure that fault-tolerant components are not hosted within the same fault-containment region.

AADL restricts interactions between components to occur only through defined interfaces referred to as *features* as shown in Table 2. Ports are the most commonly used feature and can be further classified as *data ports* for un-queued state data, *event data ports* for queued message data, and *event ports* for asynchronous events. Ports are directional, though data ports can be declared as bidirectional. *Access* declarations define accesses to data or bus components that another component may require or provide. *Subprogram* declarations define access to a subprogram that a component may require. The interactions between ports, port groups, access, and subprogram features and the component they interact with are specified by connections. *Connections* can be named and may have other attributes, such as whether data is sent immediately or is delayed.

Table 2 – AADL Feature Categories

Port	Communications interface for the directional exchange of data
Port Group	A collection of ports or other port groups
Access	Interface for direct (non-port) access to data or bus components
Subprogram	Interface for invoking and passing values to and from a subprogram

AADL makes a strong distinction between a component *type* and a component *implementation*. A component type specifies the externally visible interface to a component by listing the features of the component and their attributes. A component implementation defines the internal structure of a component type by listing its subcomponents and their connections. A component type definition cannot include subcomponents or connections, while a component implementation cannot define new features for the component. This is very similar to the concept of package specification and package body found in the Ada language on which AADL was originally based.

Some systems may require different system configurations at different times. AADL allows the designer to specify *modes* that represent alternative operational states of a system.[2] Transitions between modes can be specified using a standard state transition diagram. For each mode, one can specify different active components and connections, different calling sequences for threads, and mode-specific properties for components.

Finally, most AADL constructs are assigned a built-in set of *properties* that consist of a name, type, and value. Examples include period, worst-case execution time, deadlines, space requirements, and arrival rates. Properties can be assigned values in a specification through property-association declarations.

A collection of AADL packages and property sets representing a system specification is referred to as a *declarative model*. A declarative model defines, or declares, a library of component types and implementations that can be used to construct a system. In contrast, a system *instance model* is an instantiation of a specific system component implementation. If a declarative model is viewed as a set of blueprints on how to construct a system, an instance model is analogous to the operational physical system. Instance models are automatically

[2] AADL modes are not used in this project since dynamic reconfiguration is not used in any of the examples.

generated from a component implementation by recursively instantiating each subcomponent from its declarative specification. While some analyses are best performed on the declarative model, other analyses are best performed on an instance model.

Like SysML, AADL also provides mechanisms for extending a specification and the AADL language itself. Components can be extended and assigned new properties, features, connections, etc. New property sets can be defined and the properties within them assigned to AADL constructs. Finally, *annexes* can be defined to make major additions to the language that are recognized and checked by analysis tools.

The AADL Behavior Model Annex [15], [16] is an extension to AADL that allows component behaviors to be described as state transition systems with guards and actions. The guards and actions may refer to and modify AADL components and their features as well as local variables defined in the annex. So a transition can receive inputs from a port of the enclosing component or a subcomponent, reference the current value of a data subcomponent, bus, or local variable, send outputs to a port of the enclosing component or a subcomponent, and set the current value of a data subcomponent, bus, or local variable. Only flat, i.e., non-hierarchical, state transition systems are supported.

The AGREE Annex is an extension to AADL that allow component behaviors to be specified as assume/guarantee contracts. The AGREE environment allows a user to prove that the guarantees stated in an AADL type's contract are guaranteed by its implementation. It also checks that a subcomponent's assumptions are guaranteed by the component's assumptions and the guarantees of the other subcomponents and that the entire contract for a component and its subcomponents are consistent.

Tool support for AADL, AGREE, and the Behavior Annex is provided by the Open Source AADL Tool Environment [17]. OSATE is implemented as a number of plug-ins for the open-source Eclipse Integrated Development Environment (IDE). OSATE provides graphical and textual editors for AADL and supports a number of analysis tools.

3.1.3 The Kind SMT-based Model Checker.

There are many model-checkers, each with their own strengths and weaknesses [18]. Kind is an automated tool for checking safety properties of Lustre models [12], [19], [20], [21]. Lustre is a synchronous dataflow language which can be used as either an executable specification language or a highly declarative programming language [22]. Lustre is an open standard and is also the textual representation used by the SCADE Suite for the design of safety-critical systems [6]. It operates on *streams*, infinite sequences of values of conventional data types, such as machine integers, floating point numbers, and Booleans. A Lustre program, or *model*, is defined as the synchronous parallel composition of one or more *nodes*. A node is in essence an equational specification of a stream transformer, mapping a finite set of streams to another finite set of streams. Operationally, a node has a cyclic behavior: at each cycle *i*, it takes as input the value of each input stream at position, or *instant*, *i* and returns the value of each output stream at instant *i*. Lustre nodes can be stateful as they are allowed to access stream values from previous instants, up to a finite limit statically determined by the node itself. Checking safety properties for Lustre models can be recast as proving that certain logical predicates are *invariant*, i.e., hold in every reachable state of the model.

Lustre has a notion of clock that allows one to define streams and nodes with different clock rates. Clocks are just Boolean valued streams that can be associated with other streams. Intuitively, a stream *s* with an associated clock *c* is defined only at those instants of a basic global clock when *c* has value *true*. This effectively defines *s* to have a slower clock rate than the basic clock rate. This clock mechanism is quite powerful and more than enough to specify quasi-synchronous systems as defined in this project. At the same time, it considerably complicates the Lustre type system since stream types depend on clocks and typical stream operators with more than one argument (such as +, <, and so on) are well defined only on input streams with the same clock rate.

Kind was designed to check arbitrary quantifier-free invariants of Lustre models. It does that by encoding internally the model and the properties to be checked as logical formulas over the background theory of integer and real numbers [19]. The problem of proving invariance of a property then amounts to checking the satisfiability of a quantifier-free logical formula with respect to this background theory. This SMT problem [23], [24], [25], [26], [27], [28] is decidable and several efficient solvers exist, so that Kind can delegate the reasoning task to one of these solvers. In addition, an SMT solver can provide a model witnessing that a property is not invariant. Such a model can be translated to an execution trace of the system that is a counterexample to the property.

The main model checking algorithm used by Kind is *k-induction*, which is a strengthening of the induction principle. To prove a property to be invariant by basic induction, one has to show that the property holds in the initial states of the system and is preserved by every single-step transition. Together the initial case and the inductive step case show that the property holds in every reachable state of the system. The two cases can be checked automatically by an SMT solver by encoding them into logical formulas

If the initial case does not hold, the property is certainly not invariant. On the other hand, if the inductive step does not hold, the property may still be invariant since the inductive step might be falsified only by unreachable states. The idea of *k-induction* is to consider not only single-step transitions, but transition paths of finite length *k* by unrolling the transition relation *k* times. The initial case is extended to check if the property holds for the first *k* steps. The inductive step then checks if in every sequence of *k* transitions, where the property is satisfied for the first *k* states, the property also holds in the *k+1*-th state. Again, if both the initial case and the inductive step case hold, the property is invariant over all reachable states of the system. If the initial case is not valid, there exists a counterexample of *k* steps that violates the property; if the inductive step is not valid, the property may still be invariant.

It is important to note that *k*-induction can prove a strict superset of the properties that can be proved by induction, or in fact *k*-induction for a smaller *k*. However, there are invariant properties that cannot be proved by *k*-induction for any *k*.

The Kind model checker executes the initial case, which is called *bounded model checking* (BMC) in other contexts, in parallel with the inductive-step case. Both processes independently increment *k* until either the BMC process finds a counterexample to the property or the inductive step process proves the inductive step for some *k* for which the BMC process found no counterexamples.

Another essential component of the Kind model checker is invariant discovery, a process that guesses and tries to discover *auxiliary invariants* about the system automatically [29]. Asserting these invariants during *k*-induction can speed up the model checking process by lowering the bound *k* required to prove a property. More importantly, it can make previously unprovable properties provable by *k*-induction and thus extend Kind’s scope.

3.1.4 The Uppaal Model Checker for Timed Automata

Uppaal is an integrated tool environment for the specification, simulation and verification of real-time systems modeled as networks of timed automata [30]. A *timed automaton* is a finite automaton extended with a finite set of real-valued clocks. The clock values all increase at the same rate and clock values can be interrogated in the transition guards of each automaton, allowing transitions to be enabled or disabled based on the passage of time. Clocks can be reset to a positive time value by transitions, but cannot be changed otherwise. The addition of clock variables make timed automata useful for modeling real-time systems. Methods for checking safety and liveness properties of timed automata have been studied extensively for many years. These features make Uppaal attractive for the verification of quasi-synchronous systems.

Timed automata in Uppaal are specified as processes, where a *process* consists of process parameters, local declarations, states and transitions. Process parameters turn into process-private constants when a process is instantiated. Local declarations describe the set of private variables to which a running process has access. States correspond to the vertices of a timed automaton in graphical form. Transitions are the edges connecting these vertices. A transition specifies a source and destination state, a guard condition, an (optional) synchronization channel, and updates to private or global data. A *system* in Uppaal is the parallel composition of previously declared processes.

To facilitate analysis of timed automata, the Uppaal model checker places restrictions on comparisons and assignments to clocks. For example, a variable of type clock can only be assigned the value of an integer expression. Moreover, transition guards are limited to conjunctions of simple clock conditions (a comparison of a clock to an integer expression or another clock) or data conditions (a comparison of two integer expressions). As described in Section 3.4.3, clocks with period, jitter and offset can be readily modeled in Uppaal, and quasi-synchrony constraints can then be imposed on them. Researchers at Stony Brook University have constrained clocks in Uppaal in similar ways in previous work [31], [32].

3.2 Quasi-Synchrony

Designing and verifying systems with unbounded asynchrony can be very difficult. As discussed in [33], [34], [35], the possible interleaving of the execution of the nodes and the interactions between them grows exponentially in a fully asynchronous system, limiting the complexity of the systems that can be verified through model checking. Moreover, the synchronization of the global system state has to be accomplished through hand-shaking protocols that make no assumptions about time and are consequently hard to design. Even formally specifying the desired system properties is difficult for these sorts of systems.

Fortunately, very few actual systems exhibit completely unbounded asynchrony. In most cases, the asynchrony is bounded. A convenient way to model and verify such systems is by associating a clock (a Boolean-valued function of time) with each subsystem such that the subsystem executes one step when its clock ticks (the Boolean function becomes true) [3]. The amount of asynchrony can then be specified as constraints on the clocks. For example, in fully asynchronous systems, there are no constraints placed on the clocks and each node can execute at any time. At the other extreme, in a completely synchronous system the clocks are constrained to tick at exactly the same time so that all nodes execute in unison.

A straightforward approach to verifying such systems is to drive the local clocks with a calendar automaton. A *calendar automaton* is a discrete state machine that knows the period and jitter of each clock and maintains a global system time. At each step of the overall system, it non-deterministically computes the next time at which one or more local clocks can execute, sets their clocks true, executes the entire system, and advances the global time. It thus constrains the clocks so that nodes execute at the specified periodicity, but may still execute a bit sooner or later than a perfect clock due to their jitter. In this way, the possible interleavings of the nodes can be modeled and the overall correctness of the system verified. The primary disadvantage of this approach is that most model checkers are still overwhelmed if realistic ranges for period and jitter are used.

What is needed is an abstraction of the local clocks that makes formal verification tractable. The quasi-synchronous abstraction defined in [4] is appropriate for many real systems. This abstraction constrains the clocks so that no clock is allowed to tick more than twice before all other clocks have ticked at least once (2/1 quasi-synchrony). This is an over approximation of systems in which the clocks execute at the same period modulo their drift and jitter. In such systems, it is easy to envision a case in which one clock ticks slightly later than every other clock and then ticks again slightly before every other clock. However, it should not be possible for a clock in such a system to tick three times before every other clock has ticked at least once.

An example with three clocks is shown in Figure 2. The tick of each clock is indicated by a triangle, where time increases to the right (in both columns). The shaded background bars provide a visual frame of reference, but are not actually necessary. They are each one clock period in width. In case (a_L) all clocks tick at the same time at the start of each period. This is the synchronous case in which there is no jitter or offset. It clearly satisfies the quasi-synchronous constraint. In case (a_R), clock 2 is ticking faster than clock 1 and clock 3 and in several cases ticks three times before clock 1 or clock 3 has ticked once. One such case is indicated by the vertical dashed lines.

In example (b_L), the clocks exhibit significant jitter, but none of them ever ticks three times before all the other clocks have ticked once. It satisfies the quasi-synchronous constraint. Case (b_R) is identical to case (b_L) except that clock 2 ticks one extra time such that it ticks three times before clock 3 ticks once. Case (c_L) is similar, but still meets the quasi-synchronous constraint. Case (c_R) is identical to (c_L) except that clock 3 has ticked one extra time, causing it to tick three times before clock 2 ticks once. In case (d_L), clock 3 ticks twice as fast as clock 1 and clock 2, but never ticks so fast that it violates quasi-synchrony. Case (d_R) is identical to (d_L) except that clock 2 ticks slightly early and slightly late in one period so that clock 3 ticks three times before clock 1 and clock 2 tick once. Finally, in case (e_L) all clocks are ticking twice as slowly as expected, but never violate quasi-synchrony. Case (e_R) is identical to (e_L) except that clock 2 resumes its normal rate for one step and ticks three times before either clock 1 or clock 3 ticks once.

These examples illustrate that quasi-synchrony is clearly an over approximation of the case where all clocks execute at the same rate modulo comparatively small values for jitter. In Section 3.2.1, we quantify the relationship between period, drift, and jitter that must be maintained for the quasi-synchronous constraint to hold. So long as this relationship holds, a quasi-synchronous abstraction of a system will exhibit all the possible interleavings of the actual system. Thus, if a property of a quasi-synchronous abstraction of the system can be verified, that property will also hold of the actual system. Just as importantly, the quasi-synchronous abstraction provides a light-weight constraint on the clocks that is well suited for formal verification with model checkers.

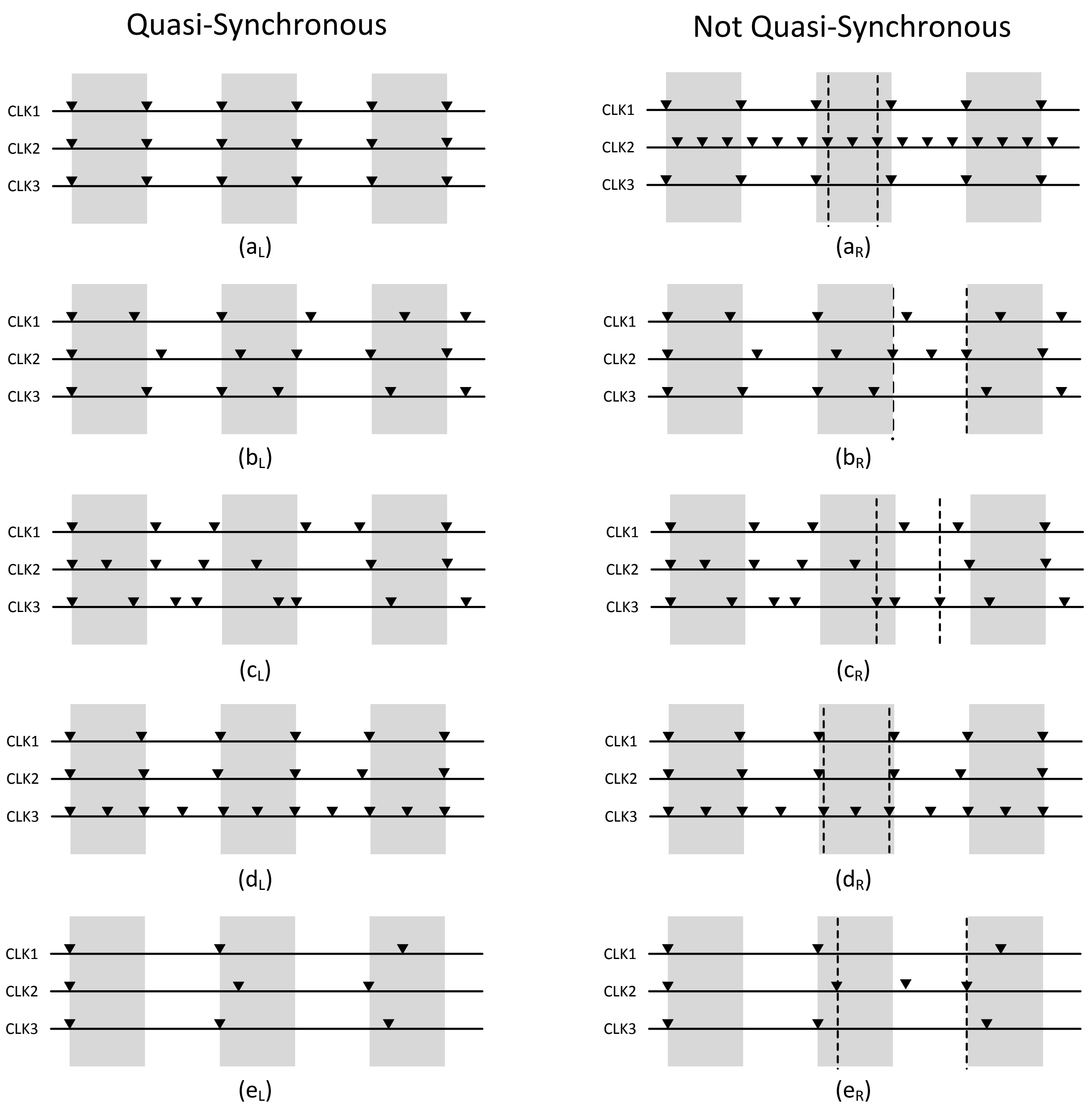


Figure 2 – Examples of Quasi-Synchrony

Of course, there are other constraints that may be of interest. For example, we could require that no clock can tick more than n+1 times before every other clock has ticked at least n times, where n is zero or greater. Another interesting constraint would be to require that no clock can tick more than n+1 times before every other clock has ticked m times, where $1 \leq m \leq n$, thereby allowing some clocks to tick slower. It is even possible to consider individual constraints between every pair of clocks. Whether such higher forms of quasi-synchrony have any practical applications appears to be an open question.

3.2.1 Relationship to Real Time.

To relate the quasi-synchronous constraint to real-time, we first characterize each clock by its period and jitter, where jitter is the maximum difference between the observed clock period and the ideal clock period. Figure 3 illustrates the relationship for two clocks *i* and *k,* with periods p_i and p_k and jitter j_i and j_k. Assume clock *i* has a shorter period than clock *k* so that clock *i* must tick three times before clock *k* ticks once to violate quasi-synchrony. Assuming Clock *k* ticks just before clock *i* ticks at time *t* and that clock *i* always ticks as early as possible, then clock *i* will tick three times by $t + 2(p_i - j_i)$, while the latest clock *k* can tick next is $p_k + j_k$. Thus, these clocks will satisfy the quasi-synchronous constraint if $2(p_i - j_i) > p_k + j_k$ and $2(p_k – j_k) > p_i + j_i$.

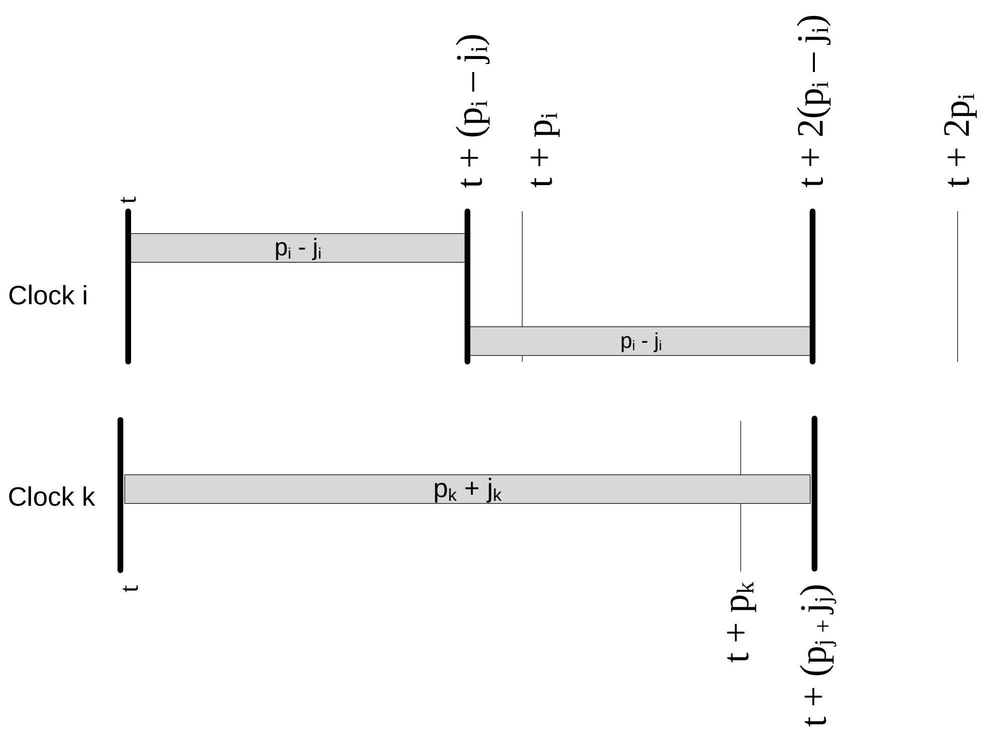


Figure 3 – Relationship of Quasi-Synchrony to Real Time

Now consider a collection of *n* clocks. Let the fastest clock be the clock *i* with the smallest value of *p - j* and let the slowest clock be the clock *k* with the largest value of *p + j*. If $2(p_i - j_i) > p_k + j_k$ holds, then it must be the case that the quasi-synchrony constraint holds for all *n (n-1)/2* pairs of clocks. So any collection of clocks is quasi-synchronous if the fastest clock *i* and the slowest clock *k* satisfies the relationship

$$2(p_i - j_i) > p_k + j_k \tag{1}$$

Verifying the correctness of a distributed agreement protocol assuming *n* quasi-synchronous clocks thus verifies the correctness of the protocol for all possible collections of *n* clocks in which the fastest and the slowest clocks satisfy the inequation (1).

3.2.2 Enforcing Quasi-Synchrony.

When modeling a distributed agreement system, it is convenient to initially model the clocks as unconstrained Boolean inputs and only later impose the appropriate constraints on them. For example, to verify a system as a fully asynchronous system, the clock inputs can simply be left unconstrained. To verify the system as a synchronous system, the clock inputs can be set to be identical.

Constraining the clocks to satisfy quasi-synchrony is more complex but an elegant approach was described in [4]. Figure 4 illustrates a deterministic finite state acceptor (DFA) that accepts all valid sequences of clock activations for a pair *P* and *Q* of quasi-synchronous clocks such that no clock can tick more than twice before every other clock has ticked at least once, i.e., 2/1 quasi-synchrony. The transitions are labeled with the clock values accepted by the transition, where ($C_P.C_Q$) denotes a tick of both clocks, ($C_P.\overline{C_Q}$) denotes a tick of only *P* and ($\overline{C_P}$.$\overline{C_Q}$) denotes a tick of neither clock. State *1Q* represents the state in which *Q* has ticked once since the last time *P* ticked, while state *2Q* represents the state in which *Q* has ticked twice since the last time *P* ticked. From state *1Q* possible activations are for neither clock to tick leading back to state *1Q*, for *P* to tick leading to state *1P* (since *P* has now ticked once since *Q* last ticked), for *Q* to tick leading to state *2Q*, and for both *P* and *Q* to tick leading back to the initial state *0*. The only acceptable possibilities from state *2Q* are for neither clock to tick leading back to state *2Q* or for *P* to tick leading to state *1P*. Similar observations hold for states *1P* and *2P*.

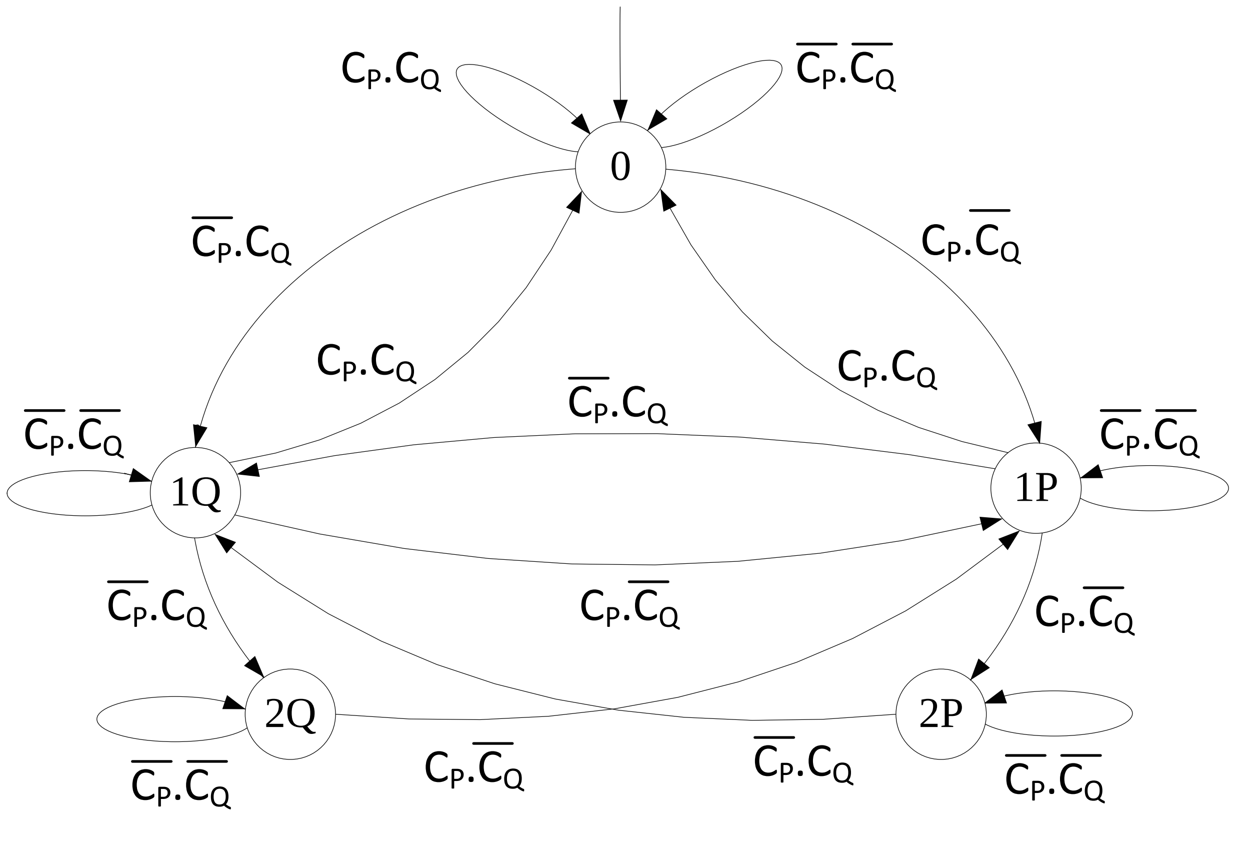


Figure 4 – Acceptor for Quasi-Synchronous Activations

For a system with *n* clocks, the DFA of Figure 4 must hold for all possible *n(n-1)/2* pairs of clocks.

3.2.2.1 Enforcing Quasi-Synchrony in Kind.

Since the Kind model checker directly accepts Lustre specifications, the clock constraints for Kind can be specified in Lustre. To constrain the clocks so that they observe 2/1 quasi-synchronously, we define the Lustre node *qs_dfa* shown in Figure 5 which implements the quasi-synchronous acceptor automaton of Figure 4.

```
node qs_dfa (p, q : bool) returns (ok : bool);
var
  r : int;
  r_is_bounded : bool;
let
  ok = not (((0 -> pre r) = 2 and p) or ((0 -> pre r) = -2 and q));

  r = if p and q then 0
      else if p then (if (0 -> pre r) < 0 then 1 else ((0 -> pre r)) + 1)
      else if q then (if (0 -> pre r) > 0 then -1 else ((0 -> pre r)) - 1)
      else (0 -> pre r);

  r_is_bounded = r <= 2 and r >= -2;

  --%PROPERTY r_is_bounded;

tel;
```

Figure 5 – Lustre Implementation of the Quasi-Synchronous Acceptor

This node takes as input two clocks *p* and *q*. The local variable *r* defines how far clock *p* is ahead of clock *q*. A positive value of *r* indicates that clock *p* has ticked *r* times since *q* last ticked while a negative value of *r* indicates clock *q* has ticked –*r* times since *p* last ticked. The negative values of *r* (-1 and -2) correspond to the states 1Q and 2Q, respectively, of the DFA in Figure 4, and the positive values of *r* (1 and 2) correspond to 1P and 2P. Since neither clock should get more than two ticks ahead of the other clock, the absolute value of r should be less than or equal to two. This is stated in the predicate *r_is_bounded* and passed to Kind as a property to be checked.

To ensure that the clock inputs actually observe the quasi-synchronous constraint, the translator also generates the node *quasi_synchronous_clocks* shown in Figure 6, which is true only if the ticks produced by every pair of clocks are accepted by *qs_dfa*. In the properties to be verified (discussed in Section 4.1) this constraint is asserted by the user in AGREE with the *synchrony : 2 command*. Kind will use this assertion to constrain the system inputs during evaluation so that only quasi-synchronous executions of the clocks are considered. In similar fashion, constraints are also generated as shown in Figure 6 for the cases in which the clocks are synchronous and asynchronous, allowing the user to easily evaluate the system under different clock constraints.

The quasi-synchronous property is a restriction on how fast an individual clock can tick, i.e., no clock can tick more than twice before every other clock has ticked at least once. However, it is also necessary to restrict how slowly each clock can tick since a system in which no clock ever ticks is not very interesting. In fact, a step in which no local clock ticks is not interesting since the system will not change state on such a step. So to ensure progress, we also require that at least one clock must tick on every step of the system. These two constraints ensure that the system makes progress and that no clock gets too far ahead of the other clocks.

```
node quasi_synchronous_clocks(
   CLK1 : bool; CLK2 : bool; CLK3 : bool; CLK4 : bool)
returns (ok : bool);
let
      ok =
            qs_dfa(CLK1, CLK3) and
            qs_dfa(CLK1, CLK2) and
            qs_dfa(CLK1, CLK4) and
            qs_dfa(CLK3, CLK2) and
            qs_dfa(CLK3, CLK4) and
            qs_dfa(CLK2, CLK4);
tel;


node synchronous_clocks(
   CLK1 : bool; CLK2 : bool; CLK3 : bool; CLK4 : bool)
returns (ok : bool);
let
      ok = (CLK1 = CLK2) and (CLK2 = CLK3) and (CLK3 = CLK4);
tel;


node asynchronous_clocks(
   CLK1 : bool; CLK3 : bool; CLK2 : bool; CLK4 : bool)
returns (ok : bool);
let
      ok = true ;
tel;
```

Figure 6 – Quasi-Synchronous, Synchronous, and Asynchronous Clock Constraints

Although 2/1 quasi-synchrony is the simplest and probably most useful in practice, we have investigated constraints for more general forms of quasi-synchrony. We say a clock *p* is *n*/*m* quasi-synchronous to a clock *q* if *p* can tick at most *n* times before *q* has ticked *m* times. The DFA of Figure 4 can be easily extended to accept valid sequences of clocks that are *n*/1 quasi-synchronous.

To enforce n/1 quasi-synchrony, we extend the DFA of Figure 4 to consist of states *iQ* and *iP* for 1 <= *i* <= *n*, as well as the state *0*. The meaning of state *iQ* is that clock *q* is *i* steps ahead of clock *p*, and vice versa for *iP*. The transitions out of states *iQ* for *i* < *n* are similar to the transitions out of state *1Q* of the DFA in Figure 4: a tick of both clocks (C_P, C_Q) leads to state *0*, a tick of clock *q* only ($\overline{C_P}$, C_Q) leads to state *(i+1)Q*, a tick of clock *p* only (C_P, $\overline{C_Q}$) leads to state *1P*, and the DFA remains in *iQ* if none of the clocks tick ($\overline{C_P}$,$\overline{C_Q}$). Transitions for state *iP* are analogous.

The transitions out of state *nQ* are similar to the transitions out of *2Q* in Figure 4: a tick of clock *p* only (C_P, $\overline{C_Q}$) leads to *1P*. Since clock *q* must not tick, (C_P, C_Q) and ($\overline{C_P}$, C_Q) are not accepted, and neither clock ticking ($\overline{C}_P$, $\overline{C}_Q$) leaves the DFA in state *nQ*. The transitions for *nP* are again analogous.

These modifications allow an implementation of the DFA as a Lustre node straightforwardly generalized to the one shown in Figure 5. We replace the value 2 in the *ok* stream with *n*, and generalize the *r_is_bounded* lemma to state the values of stream *r* are between the bounds -*n* and *n*. This generates a set of constraints that are very efficient during model checking.

To distinguish clock patters that respect *n*/*m* quasi-synchrony with *m* > 1, a different approach is needed. To construct a constraint for *n*/*m* quasi-synchrony, we consider the equivalent statement "If clock *p* ticks, then there have been at least *m* ticks of clock *q* since the *n*-th last tick of *p*." This definition can be implemented with *n* counters, respectively keeping track of the number of ticks of *q* since the last *n* ticks of *p*.

Formally, we define streams $c_q[i]$ with 1 <= *i* <= *n* to contain the number of ticks of *q* since the *i*-th last tick of *p*, and update in any state as follows. If only clock *q* ticks ($\overline{C}_P$, C_Q), then increment all $c_q[i]$. If only clock *p* ticks, set $\underline{c_q[i]}$ to the value of $c_q[i-1]$ and $c_q[0]$ to zero. If both clocks *p* and *q* tick (C_P, C_Q), combine the two previous actions by setting $c_q[i]$ to the value of $c_q[i-1] + 1$ and $c_q[0]$ to one. If no clock ticks, keep all $c_q[i]$ at their previous values.

Before the first instant of the system, we assume that *q* has ticked *m* times at each *n* last tick of *m*, such that it does not constrain the possible ticks of *p* until *p* has ticked *n* times. Therefore, we set $c_q[i]$ to *m* initially. Finally, *p* ticks *n*/*m* quasi-synchronous to *q* if on each tick of *p* the value of $c_q[n]$ >= *m*. A Lustre node implementing this constraint[3] is shown in Figure 7. It only constrains clock *p*, and thus it needs to be instantiated once for the ordered pair (*p*, *q*) and once for (*q*, *p*).

```
node  qs_c (p, q: bool; const n: int; const m: int) returns (ok: bool);
var
  c_q : int^n;
let
  c_q[i] =
    if (q and (not p)) then (m -> pre c_q[i]) + 1
    else if (p and (not q)) then (if i=0 then 0 else m -> pre c_q[i-1])
    else if (p and q)       then (if i=0 then 0 else m -> pre c_q[i-1]) + 1
                            else m -> pre c_q[i];
  ok = ((2*n) -> c_q[n]) >= m;
tel;
```

Figure 7 – Lustre Implementation of an Acceptor for n/m Quasi-Synchrony

An implementation of a quasi-synchronous constraint is required to take a sliding window approach as the one described. Since there is no explicit mention of clock periods in the abstraction, a constraint must look back at a fixed history of clock ticks. The constraint presented above maintains counters relative to clock ticks of *p*, not relative to the base clock. As such it has the advantage over conceivable other formulations that the state of the Lustre node does not change if no clock ticks. Thus, model checking algorithms can apply abstraction and

[3] For conciseness of presentation, this implementation uses arrays that currently are not widely supported in Lustre tools. Generally, this node would have to be instantiated for each concrete pair of values n and m.

compression methods. This is particularly important if there are more than two clocks in a system, such that there are instants where the state of the system changes although neither *p* nor *q* tick. The constraints generated by the acceptor of Figure 7 are not as efficient during model checking as those generated by the acceptor of Figure 5 for n/1 quasi-synchrony. For this reason, the AGREE tool will use the acceptor of Figure 5 when the *synchrony: n* command is specified and the acceptor of Figure 7 when the *synchrony: n, m* command is specified.

3.2.2.2 Enforcing Quasi-Synchrony in Uppaal

The quasi-synchronous constraints on the clocks were enforced in two different ways in Uppaal. In the first approach, we took advantage of Uppaal's ability to specify real-time and selected concrete values for the period and jitter of each clock that satisfied the constraints of Section 3.2.1. In the second approach, we constrained the clocks so that one clock ticked on each step in accordance with the quasi-synchronous constraints. The first approach has the advantage of using actual values of period and jitter that practicing engineers are comfortable with. The second approach is much closer to the way clocks are constrained in Kind and verifies the system for all combinations of period and jitter that satisfy the quasi-synchronous constraint.

In the first approach, the clock synchronization events for the top-level components are generated by clock processes, where each clock process is characterized by the clock's period *p* and jitter *j*. These values are stored in a global array indexed by the clock's id. For the system to be quasi-synchronous, these values must satisfy the constraints defined in Section 3.2.1. The template for a clock takes a clock *id* as a parameter and uses that *id* to index its values for period and jitter. The behavior associated with each clock is as shown in Figure 5.

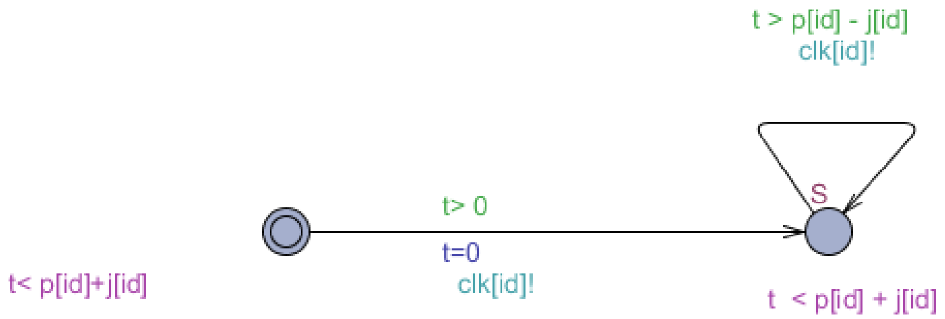


Figure 8 – Uppaal Quasi-synchronous Clocks

The offset of the clock is implemented by allowing its initial transition to occur when its local time *t* is between 0 and *p + j*, which is the latest possible time it must tick. This is implemented as the invariant *t < p[id]+j[id]* associated with the initial location. When this transition occurs, the clock transitions from the initial location to location *St_ClockTick*, generates a synchronization event *clk[id]!* and sets its local time t to zero.

After the clock transitions to the *St_ClockTick* location, the clock generates ticks in accordance with its period and jitter; i.e., a clock tick must occur in the interval (*p-j*, *p+j*). The upper transition of Figure 8 allows a tick to be generated when the time is greater than *p – j*. An invariant associated with location *St_ClockTick* forces this transition to occur before the time is greater than *p + j*.

The second approach maintains a partial history of when the clocks have ticked. At each step, one of the clocks that can tick is selected, a synchronization event is generated, and the partial history is updated. The relationship to be maintained between the clocks is specified in two arrays, *Max* and *Min* as shown in Figure 9. *Max[i, j]* specifies the maximum number of times clock *i* can tick before clock *j* has ticked *Min[j, i]* times (the values of *Max[i, i]* and *Min[i, i]* are irrelevant). Figure 9 illustrates *Max* and *Min* for 3/2 quasi-synchrony with three clocks.

Max

	Clock 1	Clock 2	Clock 3
Clock 1	▨	3	3
Clock 2	3	▨	3
Clock 3	3	3	▨

Min

	Clock 1	Clock 2	Clock 3
Clock 1	▨	2	2
Clock 2	2	▨	2
Clock 3	2	2	▨

Figure 9 – Max and Min Arrays for Clock Constraints

The partial history of clock ticks is stored in the *Age* matrix which contains one row for each clock and N columns, where N is the largest value in *Max*. Column 1 contains the number of steps since each clock last ticked, column 2 contains the number of steps for each clock since the first of the previous two ticks, column 3 contains the number of steps for each clock since the first of the previous three ticks, and so forth.

When a clock is selected to tick, every element in the row for that clock is shifted to the right (dropping the right most value), the value 0 is entered into the left most cell, and each value in the entire matrix is incremented. Initially, every value in the *Age* matrix is set to 1.[4] An example of this update cycle where clock 2 is chosen to tick is illustrated in Figure 10.

Initial Setting

	1	2	3
Clock 1	1	1	1
Clock 2	1	1	1
Clock 3	1	1	1

Clock 2 Ticks

	1	2	3
Clock 1	1	1	1
Clock 2	0	1	1
Clock 3	1	1	1

All Clocks Age

	1	2	3
Clock 1	2	2	2
Clock 2	1	2	2
Clock 3	2	2	2

Figure 10 – Update of Age Matrix when Clock 2 Ticks

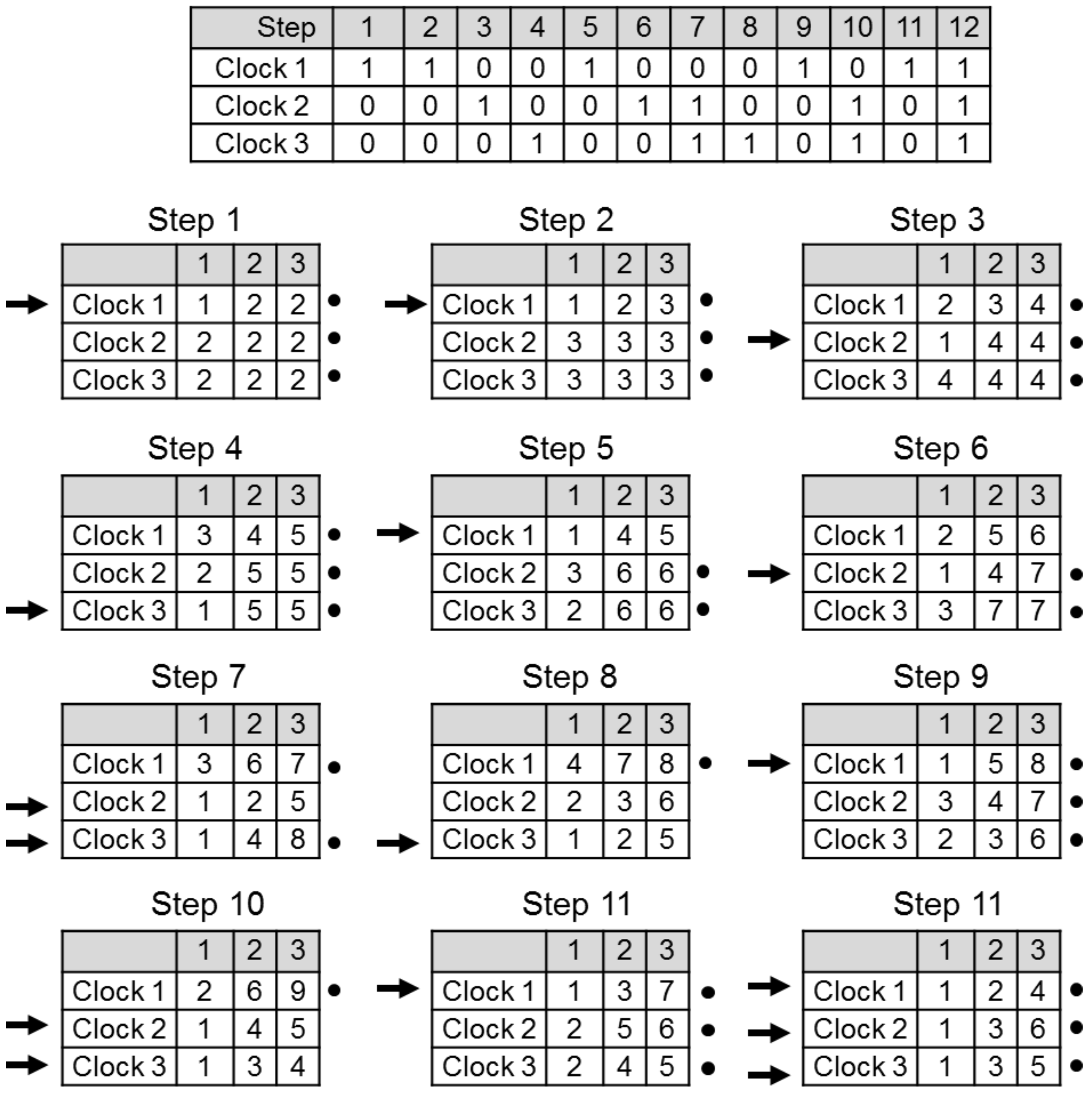
On each step, the next clock to tick can be any clock *i* that satisfies for all clocks *j*, $j \neq i$,

$$Age[i][Max[i, j]] >= Age[j][Min[j, i]]$$

An example of this is shown in Figure 11 for 3/2 quasi-synchrony. At the top of Figure 11 is a sequence of clock ticks for three clocks that satisfies 3/2 quasi-synchrony. In step 1, clock 1 ticks, in step 2 clock 1 ticks again, in step 3 clock 2 ticks, etc. Note that more than one clock can tick in a single step, as illustrated in steps 7, 10, and 11. Below are depictions of the *Age* matrix for each step. An arrow on the left indicates which clocks ticked to produce that configuration of the matrix. Dots on the right indicate which clocks are allowed to tick on the next step.

[4] Initializing the *Age* matrix with ones is not intuitive, but we have performed a proof using Kind that this initialization is correct.

For example, in step 1, clock 1 ticks, shifting its row to the right, inserting a zero in the left most cell of that row, following by incrementing the value of all cells. For 3/2 quasi-synchrony, the eligibility of each clock to tick on the next step can be determined by checking that the value of the clock in column 3 is greater than or equal to the value of all the other clocks in column 2. At the end of step 1, all three clocks are allowed to tick on the next step, indicated by the dot next to each clock.



Step	1	2	3	4	5	6	7	8	9	10	11	12
Clock 1	1	1	0	0	1	0	0	0	1	0	1	1
Clock 2	0	0	1	0	0	1	1	0	0	1	0	1
Clock 3	0	0	0	1	0	0	1	1	0	1	0	1

Step 1

		1	2	3	
➡	Clock 1	1	2	2	●
	Clock 2	2	2	2	●
	Clock 3	2	2	2	●

Step 2

		1	2	3	
➡	Clock 1	1	2	3	●
	Clock 2	3	3	3	●
	Clock 3	3	3	3	●

Step 3

		1	2	3	
	Clock 1	2	3	4	●
➡	Clock 2	1	4	4	●
	Clock 3	4	4	4	●

Step 4

		1	2	3	
	Clock 1	3	4	5	●
	Clock 2	2	5	5	●
➡	Clock 3	1	5	5	●

Step 5

		1	2	3	
➡	Clock 1	1	4	5	
	Clock 2	3	6	6	●
	Clock 3	2	6	6	●

Step 6

		1	2	3	
	Clock 1	2	5	6	
➡	Clock 2	1	4	7	●
	Clock 3	3	7	7	●

Step 7

		1	2	3	
	Clock 1	3	6	7	●
➡	Clock 2	1	2	5	
➡	Clock 3	1	4	8	●

Step 8

		1	2	3	
	Clock 1	4	7	8	●
	Clock 2	2	3	6	
➡	Clock 3	1	2	5	

Step 9

		1	2	3	
➡	Clock 1	1	5	8	●
	Clock 2	3	4	7	●
	Clock 3	2	3	6	●

Step 10

		1	2	3	
	Clock 1	2	6	9	●
➡	Clock 2	1	4	5	
➡	Clock 3	1	3	4	

Step 11

		1	2	3	
➡	Clock 1	1	3	7	●
	Clock 2	2	5	6	●
	Clock 3	2	4	5	●

Step 11

		1	2	3	
➡	Clock 1	1	2	4	●
➡	Clock 2	1	3	6	●
➡	Clock 3	1	3	5	●

Figure 11 – Example of Clock Selection for 3/2 Quasi-Synchrony

In step 2, clock 1 is again selected to tick from among the eligible clocks and the *Age* matrix is updated. At the end of step 2, all three clocks are still allowed to tick on the next step. On step 3, clock 2 is selected to tick and the *Age* matrix is updated accordingly. Step 5 is the first step in which a clock (clock 1) is not allowed to tick on the next step. This is because its value in column 3 indicates that the first of its previous three ticks occurred five steps ago, while the values for clocks 2 and 3 in column 2 indicate that the first of their previous two ticks occurred six steps ago. That is, clock 1 has ticked three times while clocks 2 and 3 have not yet ticked twice.

Clock 2 is selected to tick on step 6, which makes it possible for clocks 2 and 3 to tick on step 7. On step 7, both clocks 2 and 3 tick simultaneously, allowing clocks 1 and 3 to tick on the next step. Clock 3 is selected to tick on step 8, leaving clock 1 as the only clock that can tick on step 9. After clock 1 ticks on step 9, all three clocks are again enabled, and so forth.

The Age matrix maintains a partial history of the execution history of the clocks that is sufficient to determine which clocks can tick on each step. Column n maintains for each clock the number of steps ago that the first of its previous n steps occurred. History that is no longer needed is discarded, keeping the matrix to a fixed size. The example presented here illustrates 3/2 quasi-synchrony, but the algorithm can be extended to any form of quasi-synchrony, including forms where different constraints are specified for each pair of clocks.

The Uppaal process implementing this algorithm is shown in Figure 12. The function *checkWhichClockTicks()* determines which clocks can tick on the next step. The three branches nondeterministically select one clock to tick from among the clocks that can tick. The *performAging()* function updates the *Age* matrix, and *clk[id]!* generates the clock synchronization event just as in Figure 8.

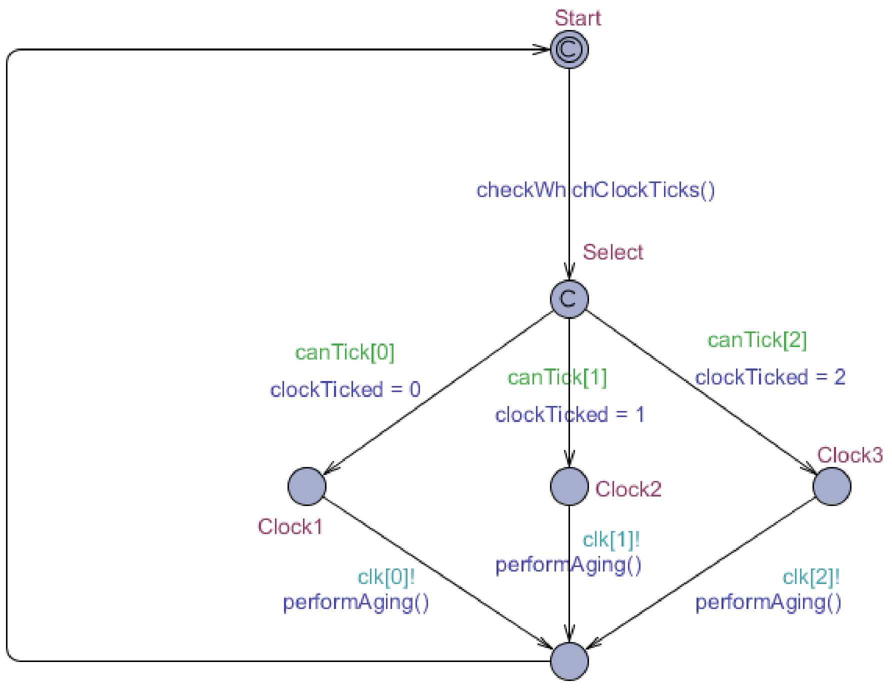


Figure 12 – Uppaal Process for Selecting Next Clock to Tick

Since only one clock is selected on each step, this actually implements a subset of true quasi-synchrony in which multiple clocks can tick on the same step. However, it does implement the maximally asynchronous form of quasi-synchrony which should be equally effective in finding errors.

3.3 Example Problems.

To illustrate the issues in the modeling and analysis of quasi-synchronous systems, we created four examples of increasing complexity, all derived from actual systems. The Pilot Flying example describes a simple system in which left and right Flight Guidance Systems (FGS) need to agree on which side is the Pilot Flying Side (PFS) of the aircraft, where either pilot may choose to transfer control to the other side at any time. It is one of the simplest distributed agreement protocols, but is still complex enough to illustrate the issues in constructing these protocols. The Leader Selection example describes a system in which N nodes report their current health to all other nodes and then select the healthiest node as the leader. It is a useful example since it can be expanded to any number of nodes. It is illustrated here with three nodes. The Active Standby example is similar to the Pilot Flying example, but bases the selection of the active side on both a pilot input and on the observed health of both sides. In addition, it allows either side to fail and heal at any time. This results in a quite complex protocol. The Wheel Braking example is derived from an accident report of a commercial air transport class aircraft that resulted in a loss of braking on landing.

3.3.1 Pilot Flying Example.

A FGS is a component of the overall Flight Control System (FCS) that compares the measured state of an aircraft (position, speed, and attitude) to the desired state and generates pitch and roll guidance commands to minimize the difference between the measured and desired state. In many aircraft, the Flight Guidance *function* at the system level is implemented as two physical sides, or channels, one on the left and one on the right side of the aircraft. These redundant implementations communicate with each other over a cross-channel bus as shown in Figure 13.

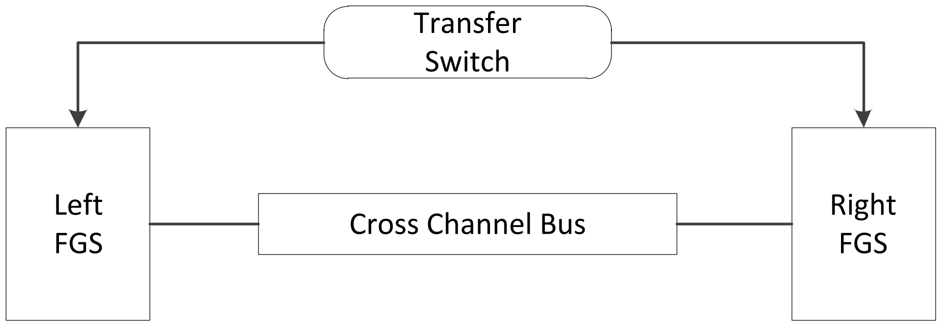


Figure 13 – FGS System Function

In most modes of operation, only one side is active and actually generating guidance commands for the aircraft. The active side is referred to as the Pilot Flying side and the other side is referred to as the Pilot Not Flying side. The flight crew can choose whether the left or the right FGS is the Pilot Flying side by pressing the Transfer Switch (TS) above the Flight Control Panel in the cockpit. If the left side is the Pilot Flying side, pressing the Transfer Switch makes the right side the Pilot Flying side, and vice-versa.

Figure 13 provides no indication of whether the two FGS execute synchronously or asynchronously. In some designs, such as a Time-Triggered Architecture (TTA), all components are driven off of a single master clock and execute synchronously. In other architectures, such as

Avionics Full-Duplex Switched Ethernet (AFDX), each component is driven by its own local clock and executes asynchronously [37]. To model a range of system architectures from fully synchronous to fully asynchronous, we introduce the more detailed model of the FGS system shown in Figure 14.

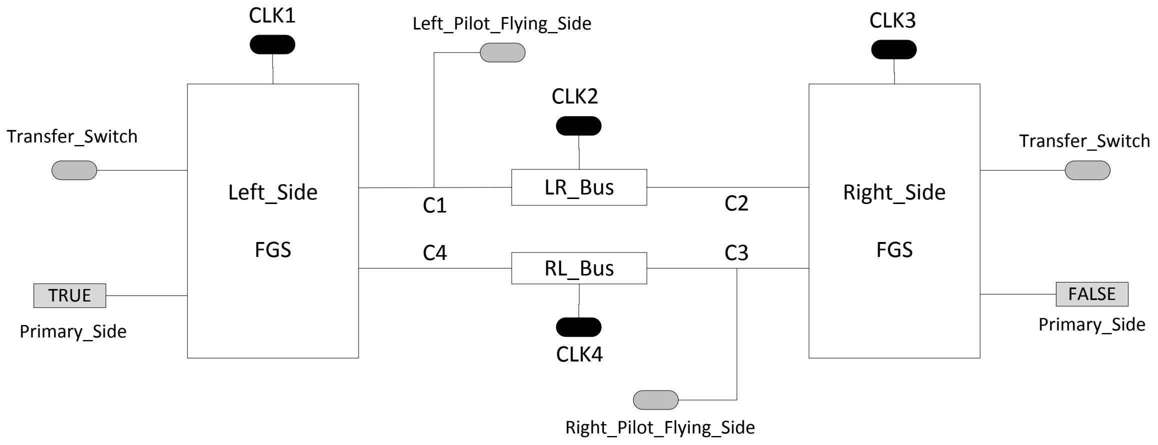


Figure 14 – Pilot Flying System Top Level Model

The FGS system of Figure 14 consists of four components - the *Left_Side* FGS, the *Right_Side* FGS, an *LR_Bus* and an *RL_Bus*. Each side produces outputs which are passed to the other side across the appropriate bus, introducing a one-step delay in the process. Among its outputs, each side produces a *Pilot_Flying_Side* Boolean output indicating if it believes itself to be the current pilot flying side. Each FGS accepts as inputs a Boolean value representing the current value of the *Transfer_Switch*[5] and the outputs passed from the other side. Each FGS is also assigned a single Boolean constant indicating if it is the *Primary_Side*. The *Primary_Side* constant for the left side is set to true while the right side is set to false.

To model system designs ranging from fully synchronous to fully asynchronous, we introduce for each component a single Boolean valued clock signal (*CLK1* though *CLK*4 in Figure 14). When a clock is true, the associated component will take a step. When a clock is false, the component makes no change to its internal state or outputs. This model assumes an underlying discrete model of time, where each component clock can tick only when the global clock ticks, but the global clock may tick at any rate and the component clocks may tick or not tick at any time the global clock ticks. By appropriately constraining the relationships between the four clocks, this model can emulate all combinations we are interested in. For example, to emulate a synchronous system, the clock signals are all equated to each other ensuring they all tick at the same time. To emulate a completely asynchronous system, the clocks are not constrained at all and may tick at any time. For a quasi-synchronous system, the clocks are constrained as discussed in Section 3.2.2.

The same informal top-level system requirements hold regardless of the clock constraints. These are:

R1 <u>At least one side is always the pilot flying side.</u>

[5] Note that there is an implied assumption that the *Transfer_Switch* arrives at both sides at the same time. The subsequent discussion makes clear why this assumption can be safely made.

R2 Both sides shall agree on the pilot flying side except while the system is switching sides.

R3 Pressing the Transfer Switch shall always change the pilot flying side except while the system is switching sides.

R4 The system shall start with the primary side as the pilot flying side.

R5 The system shall not change the pilot flying side unless the Transfer Switch is pressed or when the system is switching sides.

Verification of these properties for the quasi-synchronous system is discussed in Sections 4.1.1 and 4.2.1. The following sections describe the synchronous, asynchronous, and quasi-synchronous solutions.

3.3.1.1 The Synchronous Pilot Flying Example.

The top-level architecture of Figure 14 does not change regardless of the relationships between the component clocks, but simpler component implementations are possible for the synchronous case than for the asynchronous case. This section describes a specification adequate for the synchronous case in which all four clocks tick at the same time.

A synchronous bus simply maintains a copy of its inputs and its outputs. Since its clock ticks on every step, on each step it moves its inputs to its outputs and reads in a new set of inputs. This introduces a one-step delay in the propagation of its values.

Each FGS side executes the simple state machine shown in Figure 15 to determine which side is the current pilot flying side.

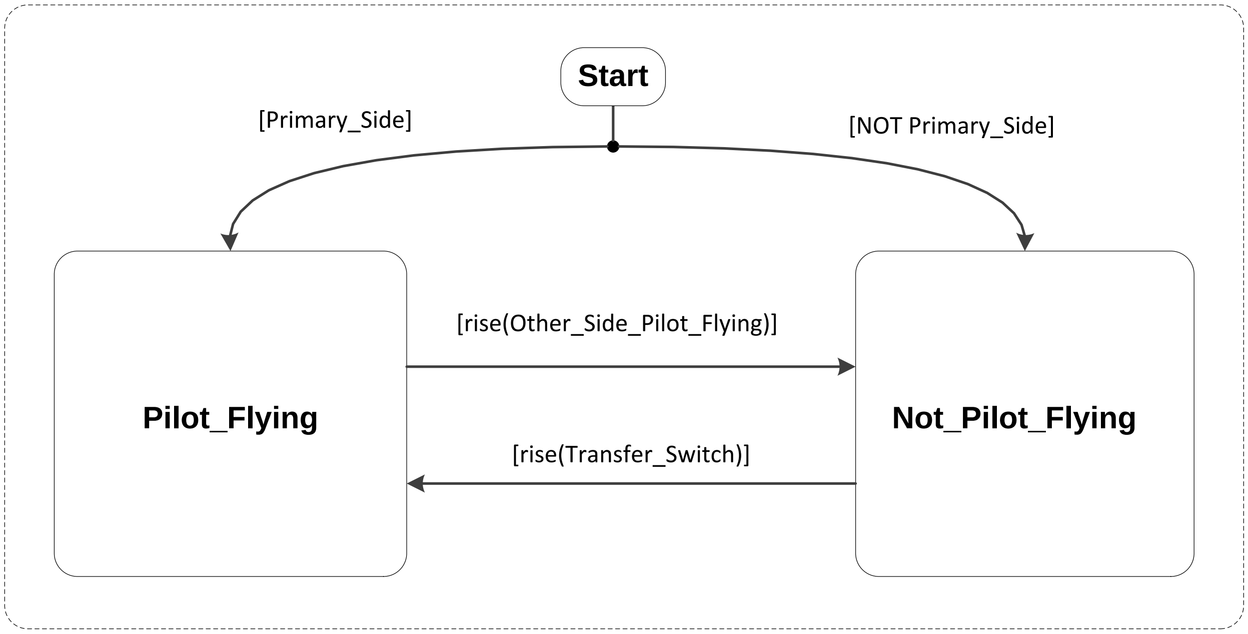


Figure 15 – Synchronous Pilot Flying Logic

If a side believes itself to be the *Not_Pilot_Flying* side, it will become the *Pilot_Flying* side when it sees the *Transfer_Switch* pressed. If it believes itself to be the *Pilot_Flying* side, it will become the *Not_Pilot_Flying* side when it sees the other side become the pilot flying side.

Thus, it is always the *Not_Pilot_Flying* side that responds to the *Transfer_Switch*, and the current *Pilot_Flying* side always yields when it sees the other side become the *Pilot_Flying* side.

3.3.1.2 The Asynchronous Pilot Flying Example

Designing and verifying the synchronization logic is more difficult in the asynchronous case when the components are not driven by a single master clock [33]. Values may be missed entirely by a component if they arrive while it is not executing, leading to race and deadlock conditions. If no assumptions are made about the individual component clocks, the Pilot Flying example can be implemented correctly only through the use of a hand-shaking protocol. The logic for this protocol is shown in Figure 16.

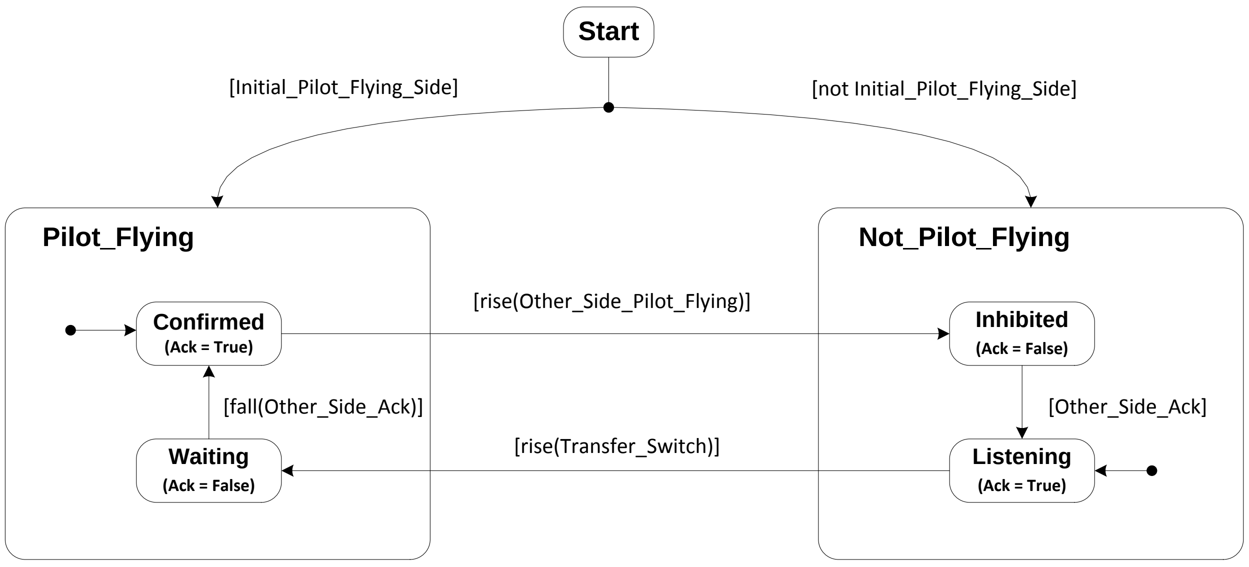


Figure 16 – Asynchronous Pilot Flying Logic

The *Ack* value is used to communicate to the other side when a side has reached a stable state. The *Primary_Side* starts in the *Confirmed* sub-state of the *Pilot_Flying* state with its *Ack* set to true. The other side starts in the *Listening* sub-state of the *Not_Pilot_Flying* state with its *Ack* set to true. When the *Transfer_Switch* is pressed, the *Not_Pilot_Flying_Side* transitions to the *Waiting* sub-state of the *Pilot_Flying* state. It remains in this sub-state until it sees the other side's *Ack* fall, indicating that it has yielded control. When the *Primary_Side* sees the other side become the pilot flying side, it transitions to the *Inhibited* sub-state of the *Not_Pilot_Flying* state and sets its *Ack* to false. Unlike the synchronous case, it does not respond to the flight crew pressing the *Transfer_Switch* while in the *Inhibited* state.[6] The *Not_Pilot_Flying_Side* transitions to the *Listening state* and begins listening for the *Transfer_Switch* when it receives an *Ack* from the other side indicating that it has reached the stable *Confirmed* state (note that it is not necessary for the *Not_Pilot_Flying* side to observe a rising edge of the *Ack*).

The only change necessary to the bus is that each message must contain two Boolean values, the *Pilot_Flying* value produced by a side and an *Ack* value used to implement the hand shake.

[6] In an actual system, this could be remedied by incorporating a delay in the delivery of the Transfer Switch longer than the time needed for synchronization of the two sides

3.3.1.3 The Quasi-Synchronous Pilot Flying Example

As might be expected, the quasi-synchronous Pilot Flying example is simpler than the asynchronous example but not as simple as the synchronous example. Since the system clocks are bounded, it is possible to use time rather than hand-shaking to implement a correct protocol. The top-level architecture of the system is identical to that shown in Figure 14. There is no need for *Ack* messages as in the asynchronous case and the cross-channel bus is identical to the bus for the synchronous example. In fact, the only change is to the logic implemented in each side. The state machine specifying the logic for the quasi-synchronous case is illustrated in Figure 17.

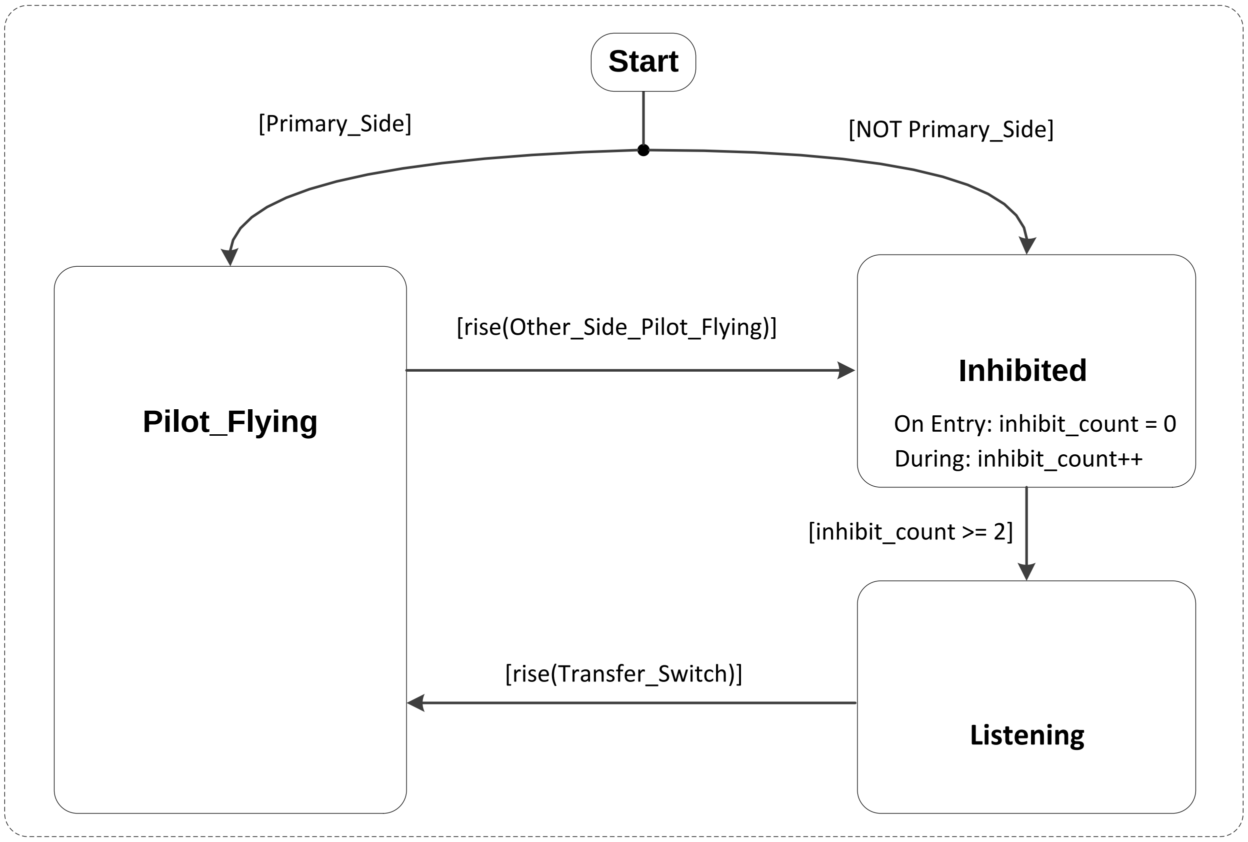


Figure 17 – Quasi-Synchronous Pilot Flying Logic

As with the synchronous case, there are *Pilot_Flying* and not *Pilot_Flying* sides, but the not *Pilot_Flying* side is implemented as *Inhibited* and *Listening* states as in the asynchronous case. The not *Pilot_Flying* side listens for the *Transfer_Switch* only when it is in the *Listening* state. After entering the *Inhibited* state, the not *Pilot_Flying* side sets the *inhibit_count* to zero, then increments it by one on each step, exiting the *Inhibited* state and entering the *Listening* state when the count becomes two. Ignoring the *Transfer_Switch* while in the *Inhibited* state is necessary to avoid the case where not *Pilot_Flying* side becomes the *Pilot_Flying* side so quickly that the current *Pilot_Flying* side fails to see the rise of the *Other_Side_Pilot_Flying* signal because its clock is low, leading both sides to deadlock as the *Pilot_Flying* side. Ignoring the *Transfer_Switch* while in the *Inhibited* state avoids this problem (just as in the asynchronous example), but uses the known bounds on the system clocks rather than handshaking to determine when to start listening again.

3.3.2 Leader Selection.

The Leader Selection example determines the healthiest node among N nodes and ensures that all nodes agree on that leader. The top-level overview for the case of three nodes is shown in Figure 18.

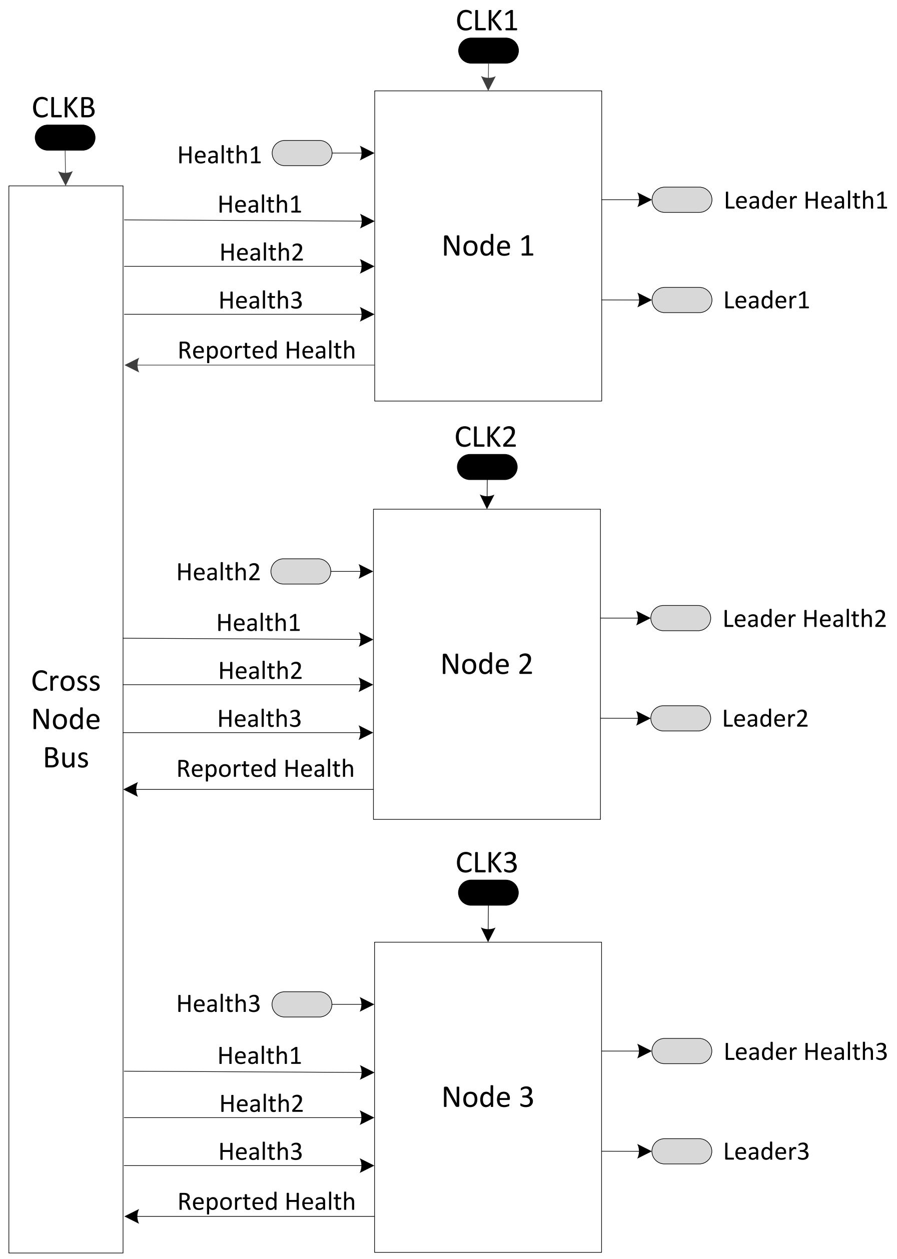


Figure 18 – Leader Selection Example with N=3

Each node computes its own health[7] on each step of its clock and communicates that to every other node (including itself) through the *Cross Node Bus*. Health is measured as an integer

[7] This is modeled simply as the input Health to that node.

from 0 to H, where 0 indicates failed and H indicates fully functioning. On each step, each node reads in the health of all nodes (including the value it sent to itself on the previous step) and selects the healthiest node as the leader. In the case of a tie, preference is given to the node with the lowest index. This is implemented as shown in the state diagram of Figure 19.

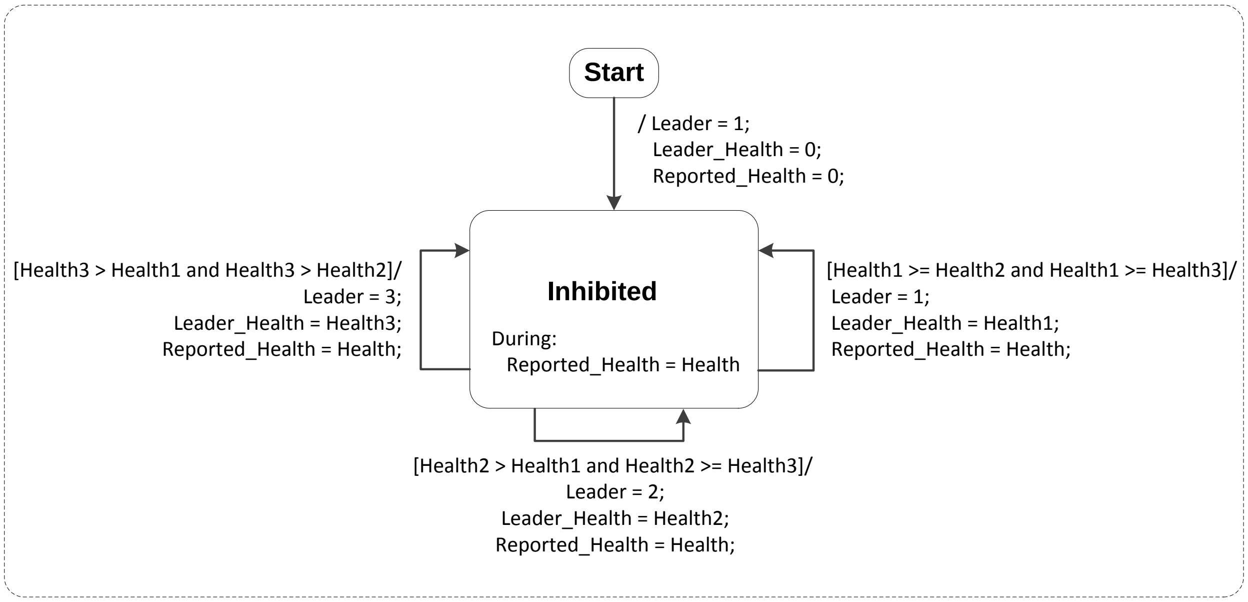

Start
/ Leader = 1;
Leader_Health = 0;
Reported_Health = 0;
Inhibited
During:
Reported_Health = Health
[Health3 > Health1 and Health3 > Health2]/
Leader = 3;
Leader_Health = Health3;
Reported_Health = Health;
[Health1 >= Health2 and Health1 >= Health3]/
Leader = 1;
Leader_Health = Health1;
Reported_Health = Health;
[Health2 > Health1 and Health2 >= Health3]/
Leader = 2;
Leader_Health = Health2;
Reported_Health = Health;

Figure 19 – Leader Selection Logic

The properties to be proven about this example are:

R1 <u>All nodes agree on the leader.</u>

R2 <u>The leader is the healthiest node.</u>

R3 <u>The leader shall not change unless the health of a node changes.</u>

Verification of these properties is discussed in Sections 4.1.2 and 4.2.1.2. The Leader Selection example is useful since it can easily be scaled to any number of nodes to study the scalability of the formal verification of quasi-synchronous systems.

3.3.3 Active Standby.

The Active Standby example determines which side (or channel) in a Primary Flight Control System (PFCS) is currently the active side. It shares many similarities to the Pilot Flying example, but it provides a much more critical function and must deal directly with the failure of either side. It also implements more complicated selection logic, monitoring the health of systems on each side of the aircraft and selecting the healthiest side as the active side, while allowing the pilot to select the active side if both sides are equally healthy. In contrast to the Pilot Flying example in which all components were modeled as AADL system components, we model the Active Standby example as both AADL system and software components. A top-level overview of the Active Standby system is shown in Figure 20.

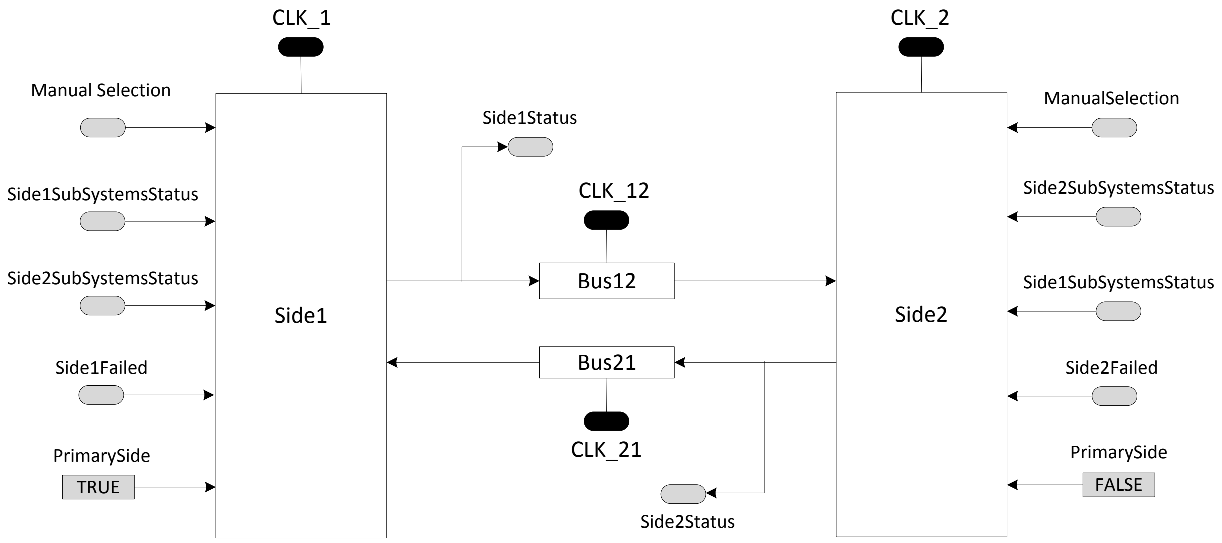


Figure 20 – Active Standby Top-Level Model

In the Active Standby example, there are two physically separate computation platforms, or cabinets, located on each side of the aircraft. These cabinets host a variety of aircraft functions including the Active Standby logic. On each side, the Active Standby logic is executed on a fail-stop processor (channel) that will declare itself as failed if it detects a failure.[8] Each side is driven by its own clock and executes asynchronously with respect to the other side.

Each side is able to sense the health of several aircraft systems located on each side of the aircraft (Side1SubsystemsStatus and Side2SubsystemsStatus) and determines if the systems on each side are fully available. The pilot has access to a Manual Selection switch that toggles the currently active side to switch to the other side if both sides are equally healthy. The failure of a side is also treated as an input to its Active Side logic. Each side communicates with the other side through a bus that reliably delivers four values to the other side in nondeterministic but bounded time:

1. that side's determination of whether it is the active side
2. that side's determination of whether it is failed
3. that side's assessment of whether the aircraft systems on that side are fully available
4. that side's assessment of whether the aircraft systems on the other side are fully available

[8] Typically implemented by two processors executing in lockstep and checking each other's computation.

The details of each side are shown in Figure 21. Since the channel is implemented in software, each component is modeled as a thread rather than as a system. The *ThisSubDemux* and *OtherSubDemux* threads unpack the status messages containing the health of each subsystem on a side into Boolean values for input to the *Monitor*.[9] The *Monitor* thread accepts the health indication of each subsystem on each side and outputs two Boolean values indicating if this side and the other side are fully available. The *ThisSideDemux* threads unpack the individual fields from the status message from the other side into Boolean values indicating whether the other side is active, the other side is failed, the monitor on the other side believes this side is fully available and the monitor on the other side believes the other side is fully available. The *ActiveSideLogic* uses all of these inputs to determine if it is the active channel. Finally, the *Mux* thread packs the individual fields from this side into a single status message to be sent to the other side.

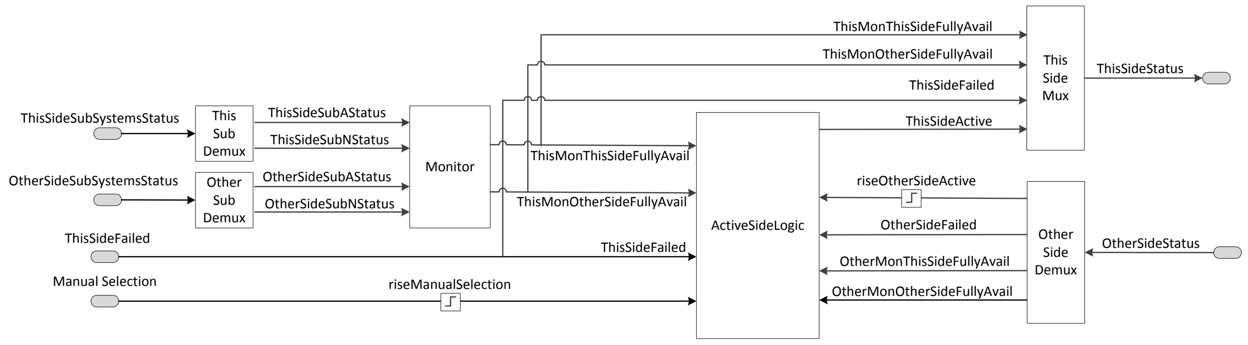


Figure 21 – Active Standby Channel Software

The logic determining the active side is shown in Figure 22. On startup, a side determines if it is failed or running based on its built-in self-test logic. If failed, it enters the *ThisSideFailed* state and sets its *Active* status to false. This will be delivered through the bus so the other side knows this side is failed. Both sides initially assume the other side is failed until they are notified otherwise.

If this side is not failed it enters the *ThisSideRunning* state and checks whether it currently believes the other side is failed. If the other side is failed (or if this side has not yet received any information from the other side), this side enters the *OtherSideFailed* state and sets its *Active* status to true (since it believes the other side to be failed). If the other side is not failed, this side enters the *BothSidesRunning* state and enters the *Init* sub-state, setting its *Active* status to whether or not it is designated as the *PrimarySide* as indicated in Figure 20. It also initializes the *InitCount* variable to zero. This counter is incremented on each subsequent state until it becomes two. This wait in the *Init* state is necessary to ensure all messages from both sides have arrived before attempting to synchronize on which side is the active side.

Once *the InitCount* reaches the threshold value of two, the side enters the *ThisSideActive* state if it is the *PrimarySide* and set its *Active* status to true. It will remain the active side until it sees the other side become the active side, at which time it enters the *Inhibit* state and sets its *Active* status to false and its *InhibitCount* to zero. Just as in the Pilot Flying example, while in the *Inhibited* state the side will ignore any inputs (such as the *Manual Selection* switch) that would cause it to become the active side, causing both sides to deadlock as the active side.

[9] Mux and Demux blocks were introduced to accommodate an early limitation of the AADL Behavior Annex that has since been removed.

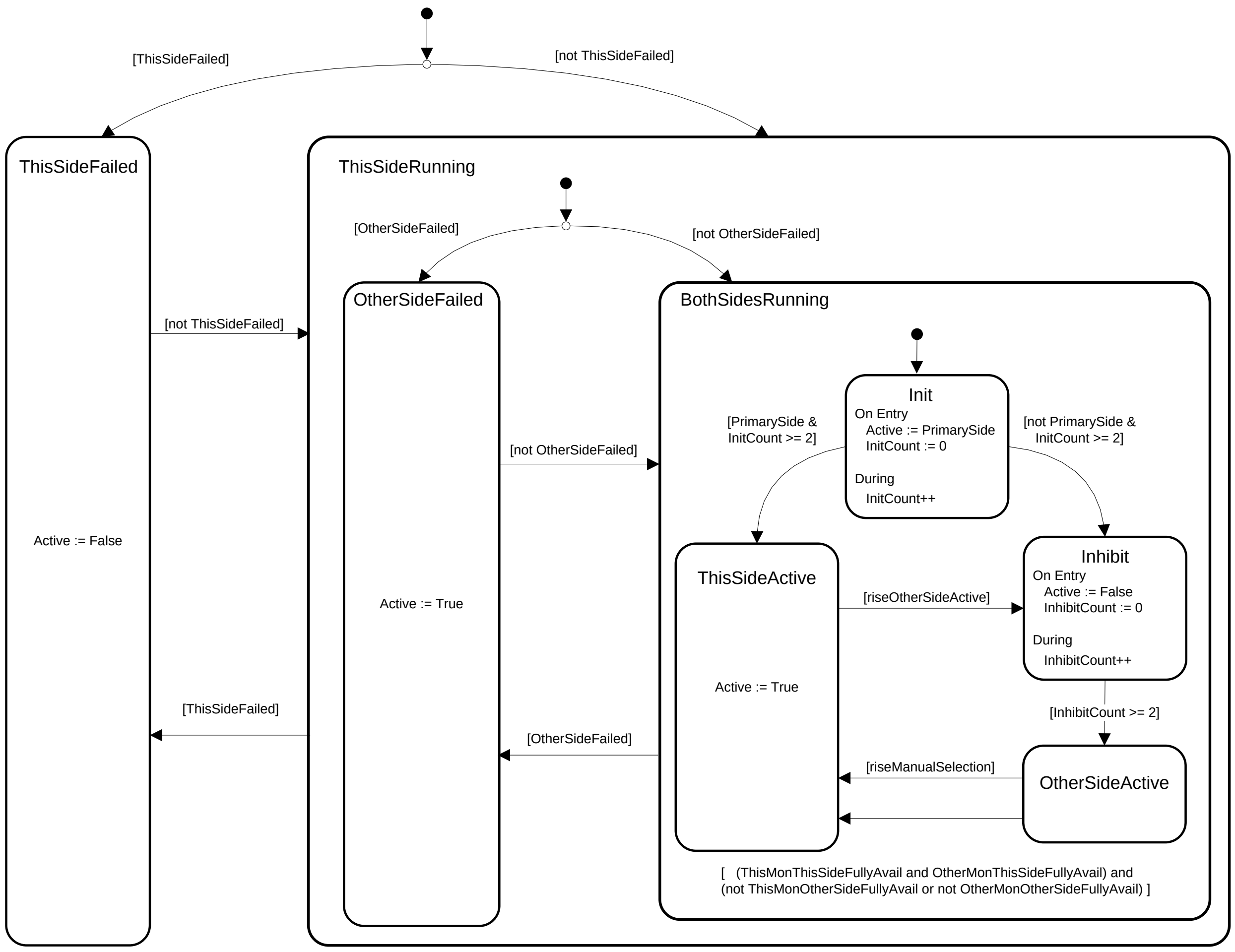


Figure 22 – Active Standby Active Side Logic

After waiting in the *Inhibit* state until the *InhibitCount* becomes two, the side enters the *OtherSideActive* state. From here the side can change its state only if a) it detects that it is failed causing a transition to *ThisSideFailed*, b) the other side reports that it is failed causing a transition to *OtherSideFailed*, c) the pilot presses the *Manual Selection* switch causing a transition to *ThisSideActive*, or d) this side determines that the other side is not fully available (not *ThisMonOtherSide2FullyAvail*) or the other side reports that it is not fully available (not *OtherMonOtherSideFullyAvail*) while both sides believe this side to be fully available (*ThisMonThisSideFullyAvail* and *OtherMonOtherSide1FullyAvail*).

There are several requirements (or properties) that should hold about the Active Standby system, including:

R0 At least one side should always be active *unless a side has just failed.*

R1 Both sides should agree on which side is active *provided neither side has failed, the availability of a side has not just changed and the pilot has not just made a manual selection.*

R2 A side that is not fully available should not be the active side if the other side is fully available *provided neither side has failed, the availability of a side has not just changed and the pilot has not just made a manual selection.*

R3 The pilot can always change the active side *except if a side is failed or the availability of a side has just changed.*

R4 If a side is failed the other side should become active *unless the other side is also failed.*

R5 The active side should not change *unless the availability of a side changes, the failed status of a side changes, or manual selection is selected by the pilot.*

Verification of these properties is discussed in Section 4.1.3.

3.3.4 Wheel Braking System.

The WBS example is drawn from the report of an Airbus A-320 accident that occurred on May 21, 1998 [38]. In this accident, an A-320 with 180 passengers and a crew of 7 lost both the Normal and Alternate braking systems on landing, forcing the pilot to steer the aircraft into a low earth embankment to avoid going through a sea wall. Though no passengers or crew were injured, the nose landing gear collapsed and the engines and nacelles suffered damage. The failure of the Alternate Braking system was attributed to the presence of frozen water and detergent. The failure of the Normal Braking system was attributed to a logic disagreement in both channels of the Brakes & Steering Control Unit (BSCU).

The BSCU (Figure 23) is a computer that controls the Normal braking, Autobrake, Nose Wheel Steering Aid and Antiskid functions. It has two physically distinct but functionally identical channels (1 and 2) which have independent power sources. The system is controlled by either of the two channels (the active channel), whichever is powered up first at start-up. If a fault develops in the active channel, the passive channel takes over. Once declared faulty, a channel cannot become non-faulty until it is serviced, so this is a non-redundant mode. Failure of both channels results in a switch from Normal braking to Alternate braking.

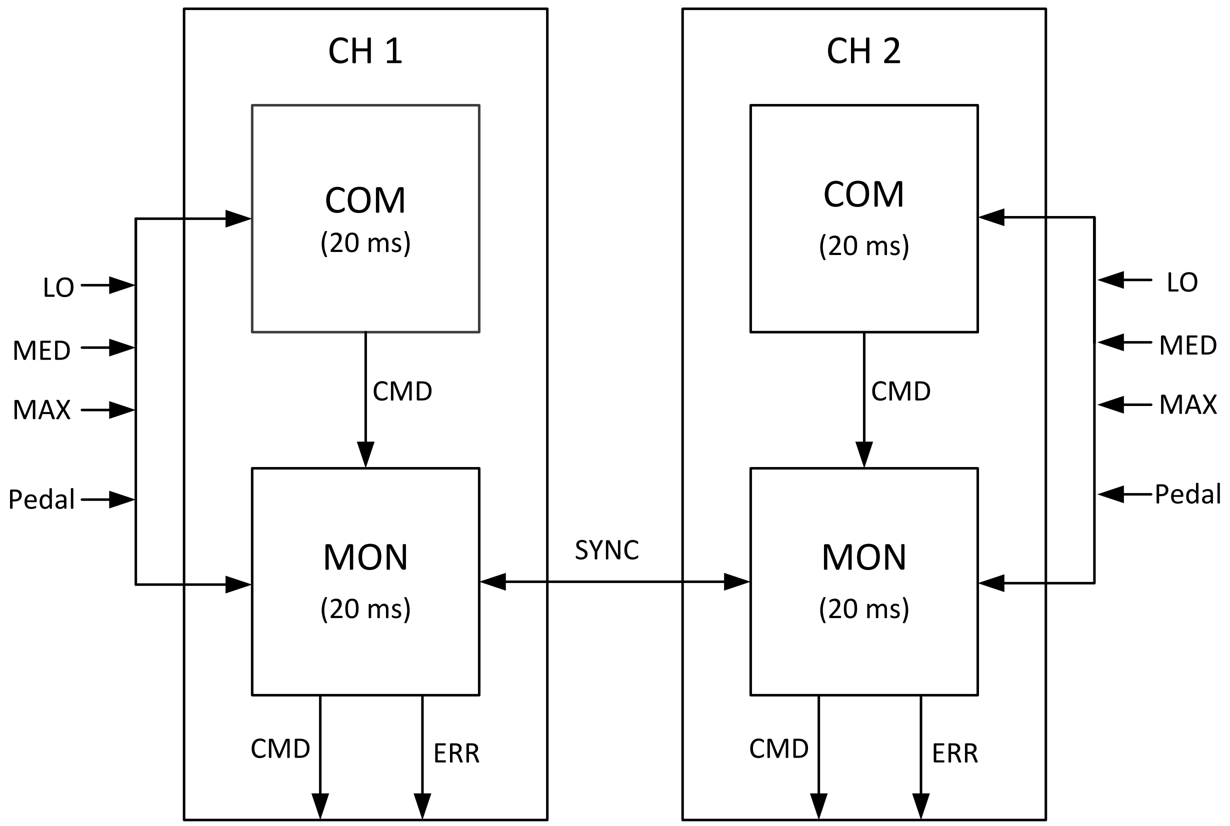


Figure 23 – Brakes & Steering Control Unit

Each of the two channels has a command function, *COM*, and monitor function, *MON*. Both the *COM* and *MON* functions compute the same output specifying the braking pressure to be applied. If the *MON* function detects a disagreement between its computed value and that of the *COM* function for six seconds, it logs a "disagree" condition within the BSCU that is also sent to the Centralized Fault Data Interface Unit. After four seconds, control is passed to the other channel.

The *COM* and *MON* units operate in four modes of operation. In *MANUAL* mode, the computed breaking pressure is determined by the pressure applied by the pilot to the brake pedal and other factors such as the Antiskid function. The pilot can also select one of three Autobrake modes in which the computed breaking pressure provides a *LO*, *MED*, or *MAX* level of deceleration. For example, in *LO* mode, the computed breaking pressure would provide 0.17g of deceleration. The Autobrake mode is selected by the pilot by pressing one of three buttons on the AUTO BRK panel. The status of the Autobrake selector pushbuttons is acquired asynchronously by the command and monitor functions every 20 ms.

To demonstrate the possible source of the BSCU logic disagreement, we create a SysML model that can be translated to AGREE and supplemented with assume/guarantee contracts. While this model is consistent with the accident report, some details must be inferred from the report and some simplifying assumptions must be made. These are pointed out in the following discussion.

The *COM* and *MON* functions respond to presses of the *LO*, *MED*, and *MAX* buttons as depicted in Figure 24. Each unit starts in the *MANUAL* mode and transitions to the *LO*, *MED*, or *MAX* mode when the associated button is pressed. Pressing the same button a second time returns the unit to the *MANUAL* mode. Pressing a different button while in an Autobrake mode selects the appropriate mode, e.g., pressing the *MAX* button while in *LO* mode selects *MAX* mode.

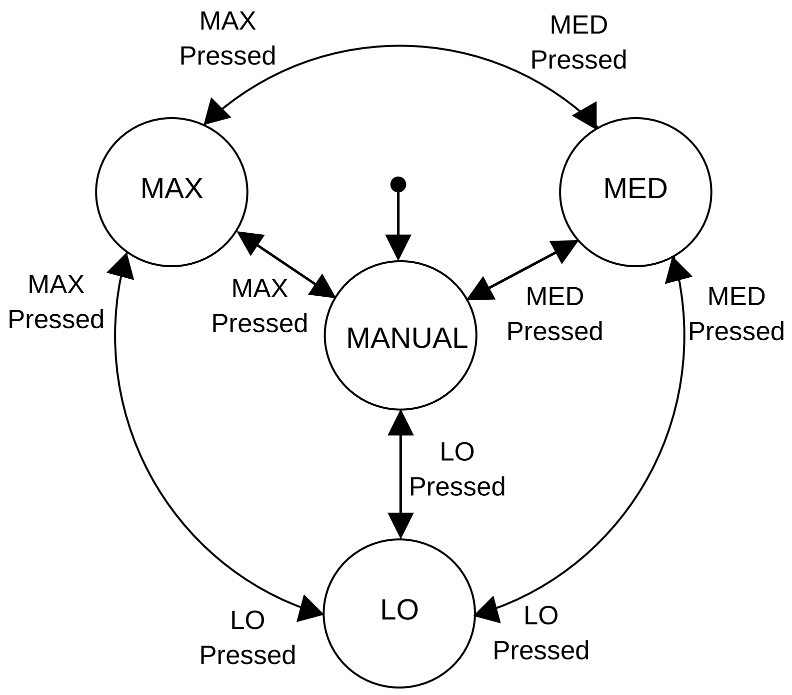


Figure 24 – COM and MON Modes

We assume that since the *COM* and *MON* functions acquire the selector pushbutton status asynchronously every 20ms, the *COM* and *MON* functions execute asynchronously with a 20ms period. Also, since channel 1 and channel 2 are physically distinct with their own power supplies, we assume they also execute asynchronously. Since the AGREE tool allows only the clocks of subcomponents of a single component to be constrained to execute quasi-synchronous[10], we eliminate channel 1 and channel 2 from the BSCU architecture, maintaining the physical isolation by restricting the communication between the channels to the synchronization logic between the *MON1* and *MON2* units as shown in Figure 25.

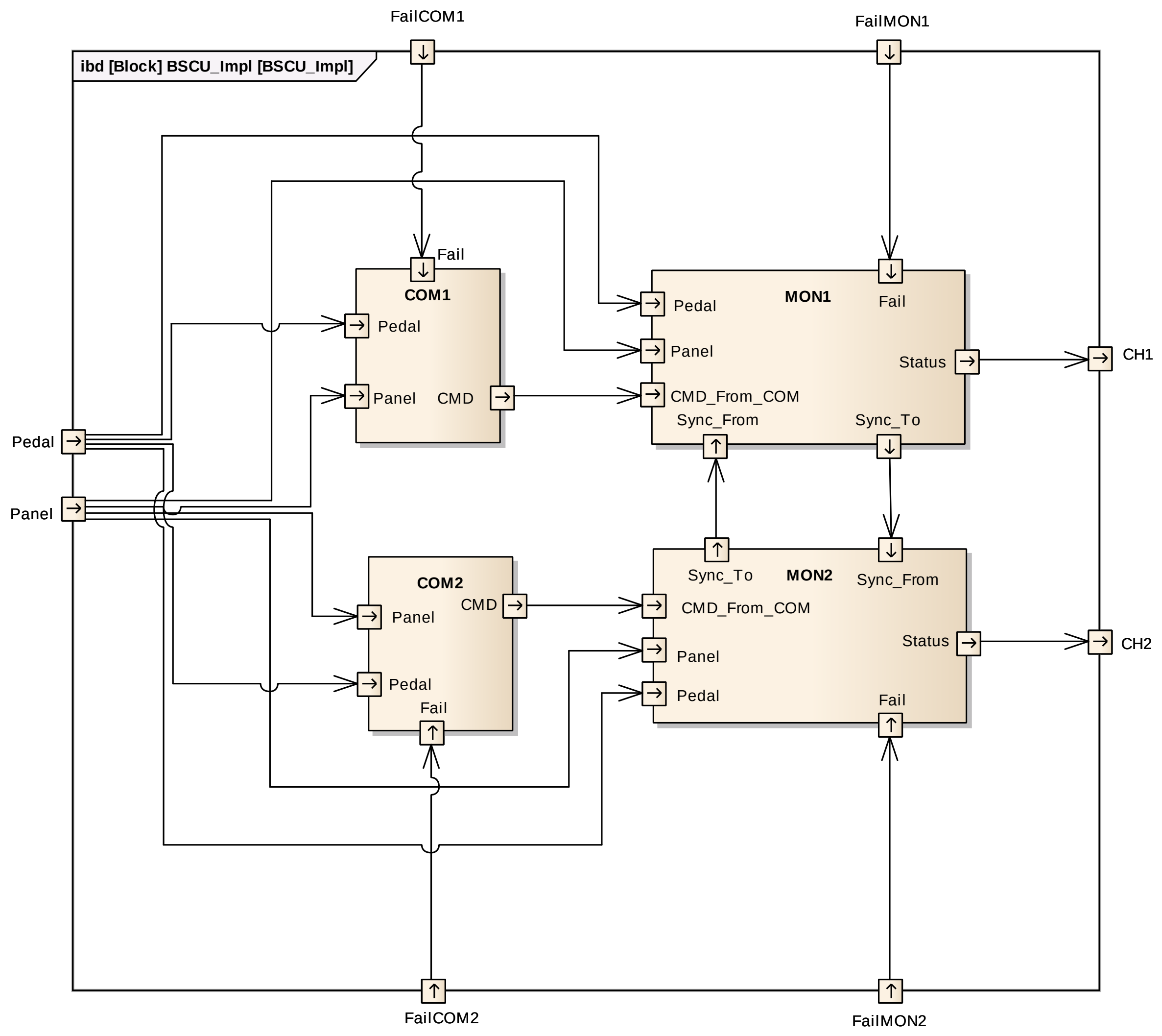


Figure 25 – BSCU Logical Architecture

This model brings in the selector pushbutton values as three Boolean values (*LO*, *MED*, and *MAX*) through the *Panel* port and the current pedal position as an integer value through the *Pedal* port. Synchronization information is exchanged between the *MON* functions of the two channels through the *Sync_To* and *Sync_From* ports. The brake command computed by the *COM* unit is passed to the *MON* unit of the same channel to be compared against the brake command computed by the *MON* unit. The output of each channel is a *Status* message containing fields *Active* (whether the channel believes it is the active side), *Error* (whether the channel believes it

[10] This restriction will be lifted in an upcoming version of AGREE.

is in error), and *Cmd* (the brake command computed by the *MON* unit). In addition, each *MON* and *COM* unit has a single Boolean input value indicating whether it is failed. This “pseudo-input” is used to simulate the failure of a unit during verification. The behavior of the *COM* and *MON* units are specified as AGREE contracts. They are discussed in Section 4.1.4 as part of the verification of the WBS example. There are six requirements (or properties) that should hold of the Wheel Braking System:

R1 At least one channel shall be error free *if no components have failed.*

R2 At least one channel shall be error free *if at most one component has failed.*

R3 At least one channel shall be active *if no components have failed.*

R4 At least one channel shall be active *if at most one component has failed.*

R5 At most one channel shall be active *if no components have failed.*

R6 At most one channel shall be active *if at most one component has failed.*

3.4 Translation of the System Models

Several translations are necessary to transform the initial SysML models into a form that can be analyzed by the Kind and Uppaal model checkers. First, SysML is translated into AADL. State machines contained within a SysML block are translated into both an AADL AGREE Annex and an AADL Behavior Annex. The AGREE Annex is used when doing verification with AGREE and the Behavior Annex is used when doing verification with Uppaal. Additional behavior can be specified as AGREE contracts which are inserted into the appropriate component during translation.

Formal verification can then be performed directly using the AGREE tool from the AADL model supplemented with AGREE contracts. The AGREE tool translates the model into the Lustre language which can be read by the Kind model checker, submits the translated Lustre file to Kind for analysis, reports on the status of the proofs, and formats counterexamples for review by the user.

Formal verification using Uppaal requires an additional translation step to generate an Uppaal model from the AADL model and the Behavior Annex specifications. The generated Uppaal mode can then be verified using the user interface provide by the Uppaal model checker.

The following sections discuss the translation from SysML to AADL, AGREE, and the Behavior Annex and from AADL to the input language of Uppaal. Additional details are available in the appendices.

3.4.1 Translating SysML to AADL.

The SysML/AADL translator translates system architectural models specified in a subset of the SysML modeling language to models specified in a subset of the AADL modeling language. Translating to AADL provides an intermediate textual format based on the SAE standard for AADL, ensures the models have been screened for many common errors that SysML does not address, and makes accessible the analysis tools already developed for AADL, while still allowing the system developers to keep their models in the SysML modeling environment. Using AADL as an intermediate step will also simplify the translation to additional verification tools in the future.

As a general rule, SysML blocks that have no internal structure are translated into AADL component types, SysML blocks that realize a SysML block through an explicit realization relationship are translated into AADL component implementations, and SysML ports and connections are translated into AADL ports and connections. If a SysML block does not realize another block and contains internal structure such as parts and internal connections, it is split into an AADL component type and an AADL component implementation.

The SysML models must be created using the Sparx Systems Enterprise Architect® tool and stored in an Enterprise Architect repository [8]. A SysML profile for AADL extends SysML with constructs commonly used in AADL. These include AADL component categories such as system, process, thread, data, device, processor, bus and memory and AADL features such as port, data access, and bus access. These constructs are provided on an AADL toolbar palette in Enterprise Architect to simplify the development of SysML models that can be translated to AADL.

The translator is invoked from within OSATE. The translator is packaged as plug-ins for the Eclipse development environment containing Java source code, executable byte code, and supporting Extensible Markup Language (XML) files. It can be installed simply by copying the plug-ins into the Eclipse drop-ins directory. Once installed, new menu items are provided to the

OSATE user to import a SysML model as an AADL model and to export an AADL model as a SysML model[11]. Users can modify and extend the translator by modifying the Java source code and recompiling the plug-ins.

Detailed instructions for creating SysML models that can be translated into AADL are provided in the SysML to AADL User's Guide included in the tool distribution.

3.4.2 Translating SysML State Machines to AGREE and to the Behavior Annex.

SysML state machines contained within a block are translated into both an AGREE Annex and a Behavior Annex contained within the corresponding AADL component type or implementation. The AGREE Annex is used when doing verification with AGREE and the Behavior Annex is used when doing verification with Uppaal. Maintaining both the AGREE Annex and the Behavior Annex does not cause any conflicts in OSATE.

As an example, the SysML state machine for the Pilot Flying logic of Figure 17 is translated into the Behavior Annex shown in Figure 26.

```
  annex behavior_specification {**

      variables
        inhibit_count: Base_Types::Integer;
      states
        St_Inhibited :  state ;
        St_Listening :  state ;
        St_Pilot_Flying :  state ;
        St_Start :  initial state ;
        St_Stop :  complete final state ;
      transitions
        T4 : St_Inhibited -[ inhibit_count >= 2 ]-> St_Listening ;
        Do_St_Inhibited :
             St_Inhibited -[ not (inhibit_count >= 2) ]-> St_Inhibited
          { inhibit_count := inhibit_count + 1 }  ;
        T6 : St_Listening -[ riseTS ]-> St_Pilot_Flying
          { PFS := true }  ;
        T3 : St_Pilot_Flying -[ riseOSPF ]-> St_Inhibited
          { PFS := false; inhibit_count := 0 }  ;
        T1 : St_Start -[ QS_Properties::Primary_Side ]-> St_Pilot_Flying
          { PFS := true }  ;
        T2 : St_Start -[ not QS_Properties::Primary_Side ]-> St_Inhibited
          { PFS := false; inhibit_count := 0 }  ;

**};
```

Figure 26 – Behavior Annex for Pilot Flying Logic State Machine

The mapping from the states of Figure 17 to the states of Figure 26 is immediate except for the prefix "St_" (which helps to prevent name clashes) and the addition of the *St_Stop* state. The Behavior Annex requires a final state which must be modeled as a Final State in SysML (the

[11] The translator from AADL to SysML is only partially implemented.

Stop state is omitted from Figure 17). If a state machine does not terminate, the *Stop* state is included in the SysML state machine, but cannot be reached from any other state.

The *inhibit_count* variable is modeled in SysML as an integer attribute of the *Inhibited* state and translated into a Behavior Annex local variable. The transitions between states are labeled with their SysML name and identify the starting state, the transition guard, and the destination state. Transition guards may reference local variables (such as the *inhibit_count*), input ports, or AADL properties. Transition actions may change local variables or set values on output ports. SysML *during* actions that occur while in a state (such as incrementing the *inhibit_count* in the *Inhibited* state) are translated into transitions to and from the same state (such as the *Do_St_Inhibited* transition in Figure 26). Note that the guard for this transition ensures that precedence is given to the exit transition *T4* when *T4* is enabled.

A portion of the AGREE Annex generated for the same Pilot Flying logic of Figure 17 is shown in Figure 27. States are translated into AGREE integer constants. The current state is translated into an integer valued stream[12] named *state* that is defined in an *eq* (equation) statement.[13] Local variables such as the *inhibit_count* are also translated into AGREE stream variables. The values these variables can assume are specified with AGREE assert statements if the enclosing component is an AADL implementation as in Figure 27 or AGREE guarantee statements if the enclosing component is an AADL type.[14] For example, the first assert statement of Figure 27 states that the *state* stream variable must equal *St_Start* in the initial state. The value of *state* on subsequent steps is not specified by this statement since the -> [15] operator specifies only that the predicate *true* must (trivially) hold in all subsequent steps. The value of *state* on steps other than the initial step is specified in other assert (or guarantee) statements. For example, the next assert statement states that the value of state on any step must be one of the allowed state constants defined earlier.

The next *eq* statement introduces a Boolean stream variable named *T4* that defines when the transition from *St_Inhibited* to *St_Listening* can occur. This transition is enabled whenever the state on the previous step was *St_Inhibited* and the previous value of the *inhibit_count* is greater than or equal to two. The next assert statement guarantees that whenever *T4* is enabled, the value of *state* is *St_Listening*. If other local variables were affected by the execution of *T4*, there would also be an assert statement defining the new value of each affected variable. For example, the Boolean variable *Do_St_Inhibited* defining when the *during* transition from *St_Inhibited* to *St_Inhibited* is enabled is followed by assert statements defining the new values for *state* and *inhibit_count*.

These definitions define the new values of state and local variables when a transition is taken. Since AGREE is constraint based specification language, it is also necessary to define

---

[12] AGREE variables are streams, which are mappings from an execution step 0, 1, 2 … to the value of the variable on that step, similar to variables in the Lustre language [22].

[13] AGREE variables are defined in equation (eq) statements similar to the style of PSL [36].

[14] The AGREE contract for an AADL type specifies the behavior any instance of that and only guarantees are allowed in the contract for an AADL type. The contract for an AADL implementation specifies how the subcomponents of the implementation satisfy the contract for the type and can only contain assertions (i.e., statements of fact) and lemmas that can be proved about the subcomponents.

[15] The followed-by operator -> is an infix operator whose left hand expression specifies the initial value of stream and whose right hand expression specifies its value in all subsequent steps, just as in the Lustre Language [22].

their value when it is not changed by a transition. The last three assert statements do this for the output *PFS* and for the variables *state* and *inhibit_count*.

```
    annex agree {**

        const St_Inhibited : int = 1;
        const St_Listening : int = 2;
        const St_Pilot_Flying : int = 3;
        const St_Start : int = 4;
        const St_Stop : int = 5;

        eq state : int;
        eq inhibit_count: int;

        assert (state = St_Start) -> true;

        assert (state = St_Start) or (state = St_Inhibited) or
               (state = St_Listening) or (state = St_Pilot_Flying) or
               (state = St_Stop);

        eq T4: bool =
             (false -> (pre(state) = St_Inhibited) and pre(inhibit_count) >= 2);

        assert T4 => (state = St_Listening);

        eq Do_St_Inhibited: bool =
             (false -> (pre(state) = St_Inhibited and not (T4)));

        assert Do_St_Inhibited => (state = St_Inhibited);
        assert Do_St_Inhibited => (inhibit_count = pre(inhibit_count) + 1);

                  • • •                      • • •

        assert true -> (not (T6 or T1 or T3 or T2) => (PFS = pre(PFS)));
        assert true -> (not (T6 or T4 or Do_St_Inhibited or T1 or T3 or T2) =>
             (state = pre(state)));
        assert true -> (not (Do_St_Inhibited or T3 or T2) =>
             (inhibit_count = pre(inhibit_count)));
 **};
```

Figure 27 – AGREE Annex for Pilot Flying Logic State Machine

If a SysML state machine has hierarchical states such as in the Active Standby logic of Figure 22, the state machine is first flattened into a state machine without hierarchy. One state is generated for each leaf-level state in the original machine and transitions are generated from each source state to each destination state in the flattened state machine. The guards and actions are accumulated on these transitions in the same order they would be evaluated in the hierarchical state machine. However, since the evaluation of guards and actions in the original hierarchical state machine are interleaved when making a transition from a state at one level to a state at a different level (i.e., the action at one level will be performed before the guard at the next level), the flattened state machine may not perfectly preserve the semantics of the hierarchical state machine. For example, in Figure 22 if the transition to the *Init* sub-state of the *BothSidesRunning* state included a guard that interrogated a variable set by an action on a transition entering the *BothSidesRunning* state, the flattened state machine would likely behave differently than the hierarchical state machine since the corresponding check in the flattened state machine would be made before the value of the variable was set. For this reason, guards in a hierarchical state machine should not depend on actions performed by their parent states during the same transition.

3.4.3 Translating AADL to Uppaal.

The primary advantage of Uppaal over Kind is that it provides explicit support for time. Uppaal supports the creation of global and local clocks that can be set to positive integer values. This allows models and properties to be specified with references to durations so that real-time properties can be specified and verified. At the same time, several issues complicated the translation from AADL to Uppaal. This section first describes how the translation was implemented, then discusses some of these issues.

Models can be specified in Uppaal in either a textual .xta format or as an .xml file. The AADL to Uppaal translator generates an .xml file. Each AADL component implementation (or component type if an implementation is not defined) is translated into an Uppaal process *template*, which can be thought of as a parameterized process definition. Each process template specifies the name of the template, the template's parameters, and the behavior associated with the process. For example, the template name and parameters generated for the Side and Cross Channel Bus in the Pilot Flying example are shown in Figure 28.

```
Side (int id, bool &TS, bool &OSPF, bool &PFS, int riseTS, int riseOSPF, int PFSL, int clk_id)
Cross_Channel_Bus( int id, const bool Init_Bool, bool &I, bool &O, bool &prev_I,  int clk_id)
```

Figure 28 – Uppaal Process Definitions

By convention, the first parameter of each template is an integer id that will be assigned to the process when it is instantiated. This id should be unique for each instantiation of the template. Inputs and outputs of the process are specified as pass-by-reference parameters, which are prefixed in Uppaal by "&". For example, in Figure 28, the inputs and outputs of the *Side* process are *&TS*, *&OSPF*, and *&PFS*.

Since Uppaal does not allow processes to be nested, the process id of each AADL subcomponent is specified as a pass-by-value parameter so that the correct instance of the process can be referenced in the template. For example, in Figure 28 the process ids of the *riseTS*, *riseOSPF*, and *PFSL* AADL subcomponents are specified as parameters with those names. AADL properties may also be passed as template parameters. In Figure 28, the initial output of the *Cross_Channel_Bus* is specified as the constant parameter *Init_Bool*. Finally, since each AADL component instance is associated with a unique local clock, the integer id of that clock is specified as the last template parameter.

Each AADL component instance is then translated to Uppaal as an instantiation of its corresponding template with the appropriate input parameters. For example, the instantiations of the four top-level subcomponents of the Pilot Flying system are shown in Figure 29.

```
LS = Side (0, TS, RL_O, LPFS, 0, 1, 0, 2);
LR = Cross_Channel_Bus(0, true, LPFS, LR_O, LR_prev_I, 0);
RS = Side(1, TS, LR_O, RPFS, 2, 3, 1, 3);
RL = Cross_Channel_Bus(1, false, RPFS, RL_O, RL_prev_I, 1);
```

Figure 29 – Uppaal Instantiations of Pilot Flying Top-Level Processes

The *LS* process corresponding to the *Left_Side* component is assigned an instance id of 0. Inputs *TS* and *RL_O* and output *LPFS* are passed by reference, and instance id 0 for the *riseTS* process, 1 for the *riseOSPF* process, and 0 of the *PFSL* process are passed by value. Finally, the process will be clocked by clock 2. Note that processes *LS* and *RS* cannot both have the same process id, but *LS* and *LR* may since they are based on different templates.

Since Uppaal does not allow a process to directly refer to the outputs of another process, connections are made through the use of global variables. The system inputs and outputs are defined as global variables and connections between system inputs and subcomponents are made by using the system input as an argument in the process instantiation. For example, in Figure 29, the system input *TS* is passed-by-reference as an argument to both process *LS* and *RS*. Connections from the output of a subcomponent to another subcomponent are made by defining a global variable for the output port and passing that variable by reference to both processes. For example, the *RL* process is passed the global variable *RL_O*, which it uses for its output, and the same global variable is passed to the *LS* process as an input argument.

The definition of these global variables is illustrated in Figure 30. The system inputs and outputs are defined first. Note that the initial values of output ports are also provided. Following that, global variables are defined for the output ports or internal values of each process instance.

```
/*****Input and output****/
bool TS;
bool LPFS = true;
bool RPFS = false;
/**** Internal ports for pilot flying system****/
bool LR_O = true;
bool RL_O = false;
bool LR_prev_I = true;
bool RL_prev_I = false;
/**** Internal ports for Side LS****/
bool LS_riseTS_O;
bool LS_riseOSPF_O;
```

Figure 30 – Ports Declared as Uppaal Global Variables

The behavior of each process is also defined in its template. For example, the behavior of the *Cross_Channel_Bus* is shown in Figure 31. The process begins in location (state) *St_Start* by setting the value of its output port *O* (passed as the argument *&O*) to the value of *prev_I*, storing the current value from its input port *I* into *prev_I*, and transitioning to location *St_Step*. From location *St_Step* it repeats this behavior each time it receives a synchronization event *clk* from its local clock *clk_id*.

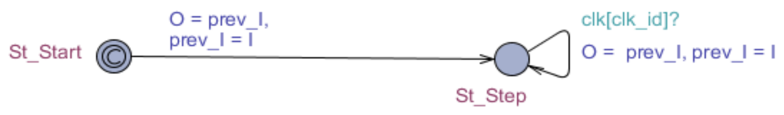


Figure 31 – Uppaal Specification of Cross Channel Bus Behavior

The processes for the top-level components are constrained to execute quasi-synchronously by the synchronization event from their local clocks, which are in turn constrained as described in Section 3.2.2.2. However, the subcomponents within a top-level component need to execute synchronously and a very different approach must be taken. Just as for the top-level components, we translate AADL subcomponent instances into Uppaal process instances. Since Uppaal processes execute asynchronously, the subcomponent instances must be constrained to execute synchronously. This is accomplished by having a component recursively invoke each of its subcomponents in the proper sequence. In this way, the top-level components are executed quasi-synchronously, but all of their subcomponents are executed synchronously in the proper execution order. To illustrate this approach, we show how the synchronization is managed for the top-level *Side* component of the Pilot Flying example. The internal structure of the *Side* component is shown in Figure 32.

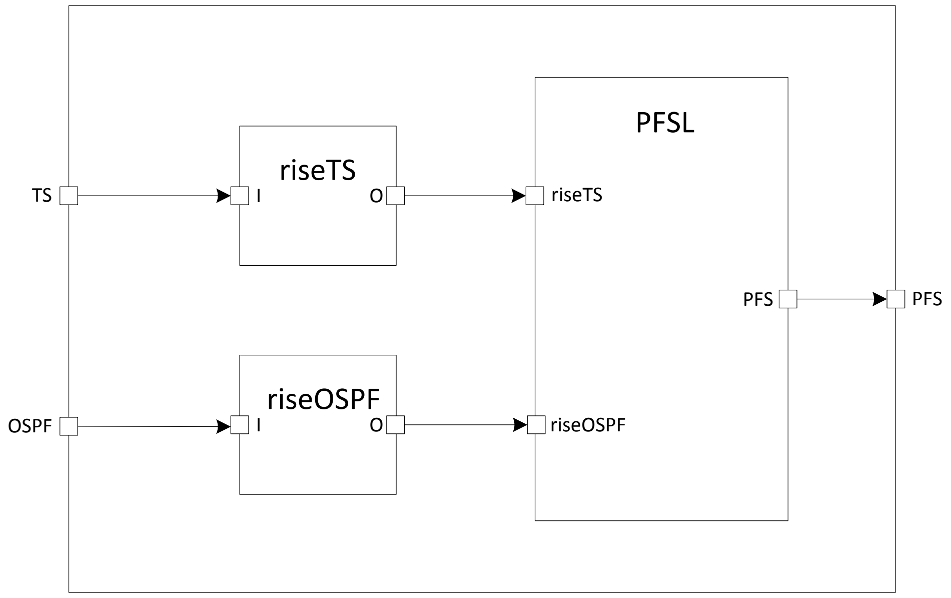


Figure 32 – SysML Internal Block Diagram for the Pilot Flying Side

Each *Side* has three subcomponents. Subcomponent *riseTS* detects a rising edge of the Transfer Switch. Subcomponent *riseOSPF* detects a rising edge of the pilot flying output from the other side (after being transmitted across the bus). Subcomponent *PFSL* uses these inputs to compute the pilot flying status of this side. These subcomponents execute synchronously with *riseTS* and *riseOSPF* executing first and providing their inputs to *PFSL*. To force the synchronous execution of these three subcomponents, the *Side* process executes the state machine shown in Figure 33.

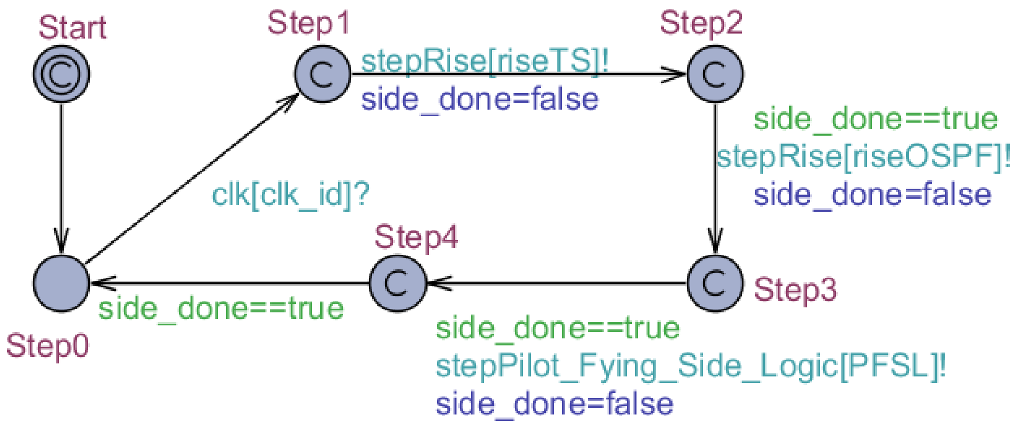


Figure 33 – Synchronizing Subcomponent Execution in Uppaal

In reading Figure 33, recall that the Side process is passed arguments *riseTS*, *riseOSPF*, and *PFSL* identifying the subcomponent processes with which it is associated. It also is passed an argument *id* identifying its own process identifier. The execution of the *Side* begins in location *Start*, but since this is an Uppaal committed state, it immediately transitions to location *Step0*. Execution of a "step" for the *Side* process starts from location *Step0* when it receives a *clk* synchronization event allowing it to transition to location *Step1*. This event is generated in the parent process for the entire Pilot Flying system in response to a tick of the quasi-synchronous clock associated with this *Side*. From *Step1*, the *Side* process issues a *stepRise* event for the *riseTS* process and sets the *side_done* semaphore to false. This allows the *riseTS* process to execute its next step, setting the *side_done* semaphore to true when it completes. The *Side* process then transitions to location *Step3*, issuing a *stepRise* event to allow the *riseOSPF* process to execute its next step. When *riseOSPF* completes, it sets the *side_done* semaphore true, allowing the *Side* process to enable the *PFSL* process and transition to *Step4*. When the *PFSL* process completes its step it sets the *side_done* semaphore true, allowing the *Side* process to transition from *Step4* to *Step0*. In this way, synchronization flows downward from each component to its child components.

Determining the execution order of subcomponents is not an issue in the translation from AADL to Kind, since the semantics of the Lustre language specify that a set of equations is evaluated in dependency order. In the above example, Kind will determine that *PFSL* depends on *riseTS* and *riseOSPF* and will pick a sequence in which they are evaluated first. It does not matter whether *riseTS* or *riseOSP*F executes first since no dependency exists between them.

Determining the execution order of the subcomponents in Uppaal requires a similar dependency analysis to be performed. In the current implementation of the AADL to Uppaal translator, we require that the designer specify the execution order of each subcomponent by attaching an *Execution_Order* AADL property to the subcomponent. The translator then uses that property to generate the synchronization state machine such as that shown in Figure 33.

As can be seen from the preceding discussion, several factors complicated the translation from AADL to Uppaal. One of the most difficult issues arose from the fact that Uppaal's main construct for modularity, the process, is also its main construct for concurrency, since all processes execute asynchronously unless constrained otherwise. Since the systems of interest here are predominantly synchronous with some asynchrony, the subcomponent processes had to be explicitly scheduled to execute synchronously by their parent processes. Since Uppaal does not determine the dependency order of subcomponents as Kind does, the determination of the execution order of the subcomponents also has to be performed by the translator.

Another issue arose from the fact that Uppaal processes cannot be nested within processes. This required the *ids* of the subcomponents of a process to be passed as parameters to that process. Finally, since an Uppaal process cannot directly reference the outputs of other processes, outputs had to be declared as global variables that were passed by reference to the connected processes.

4.0 RESULTS AND DISCUSSION

This section discusses the formal verification of the Pilot Flying, Leader Selection, Active Standby, and WBS examples. Verification with AGREE using the Kind model checker is discussed in Section 4.1and verification with Uppaal is discussed in Section 4.2.

4.1 Formal Verification with AGREE and Kind

The first step in verifying the correctness of the quasi-synchronous examples with the AGREE and the Kind model checker is to state the system requirements described in Section 3.3 as formal properties. This is done by creating an AGREE annex for the top-level system component specifying the properties as guarantees to be provided by the system.

4.1.1 Pilot Flying Verification with AGREE and Kind.

The AGREE contract specifying the formal requirements for the Pilot Flying example described in Section 3.3.1 is shown in Figure 34. This contract is associated with the AADL system block for the entire Pilot Flying example. The Boolean variable *initializing* is true when the system is starting up. It is introduced here so that we can exclude verification of properties during initialization when they may not hold. While referenced here, it is specified in contract for the *implementation* of the Pilot Flying system and will be explained later. The next two statements introduce assumptions about the Transfer Switch necessary for property R3 to hold. They will be discussed along with the verification of property R3.

The first property, R1 states that at least one side is always the pilot flying side. This is specified as the guarantee that either the output *LPFS* or *RPFS* is always true.

Property R2 states that both sides shall agree on which is the Pilot Flying side except while the system is switching sides. Ideally, both sides would always agree on which side was the Pilot Flying side, but this is not the case. The example is designed so that immediately after the Transfer Switch is pressed, the not pilot flying side becomes the pilot flying side and conveys this information across the bus to the other side. Until the other side responds and becomes the not pilot flying side, both sides will behave as the pilot flying side. Such transitory system states are unavoidable in systems where change has to propagate across the system and intermediate states are externally observable. However, if the system is not subjected to additional stimuli, it should eventually reach a stable state where no component is changing. We refer to such stable system states as *quiescent* states.

If the duration of a transitory state is short, it is often the case that a safety property only needs to hold during a quiescent state. For example, in the Pilot Flying example, we are willing to accept a transitory state in which both sides are the pilot flying side since the inertia of the aircraft can tolerate this situation for a short period of time. Property R2 makes use of the *PRESSED(p)* function, which is true if p was false on the previous step and true on the current step, and the *Duration(p)* function, which returns the number of steps *p* has been true. R2 states that if the system has not been initializing and the Transfer Switch has not been pressed for previous 25 steps[16], either the left side or the right side, but not both, will be the pilot flying side. Since this holds for all possible system states, it ensures that exactly one side will be the pilot flying side except during the transitory period when the system is switching sides.

[16] The value of 25 is discovered through experimentation with the model checker.

```
eq Initializing : bool;

-- Transfer Switch presses must be long enough to be seen by both sides
assume "No Short Presses": Agree_Nodes.True_At_Least(TS, 7);
assume "No Rapid Presses": Agree_Nodes.True_At_Least(not TS, 7);

----------------------------------------------------------------------------------
-- R1. At least one side shall always be the pilot flying side.
----------------------------------------------------------------------------------
guarantee "At Least One Side Pilot Flying" : (LPFS or RPFS);

----------------------------------------------------------------------------------
-- R2. Both sides shall agree on the pilot flying side
-- except while the system is switching sides.
----------------------------------------------------------------------------------
guarantee "Agree On Pilot Flying Side" :
   (Agree_Nodes.Duration(not Initializing and not PRESSED(TS)) > 24 =>
      (LPFS = not RPFS));

----------------------------------------------------------------------------------
-- R3. The pilot can always change the pilot flying side
--     except while the system is switching sides.
---------------------------------------------------------------------------------
guarantee "Pilot Can Change Active Side" :
   Agree_Nodes.Since(Agree_Nodes.Duration(
    pre(not Initializing and not PRESSED(TS))) > 46 and PRESSED(TS)) = 7 =>
   Agree_Nodes.Within(Agree_Nodes.Rise(LPFS) or Agree_Nodes.Rise(RPFS), 7);

----------------------------------------------------------------------------------
-- R4. The system shall start with the left side as the pilot flying side.
----------------------------------------------------------------------------------
guarantee "Left Side Initial Pilot Flying Side" :
   (LPFS -> true) and ((not RPFS) -> true);

----------------------------------------------------------------------------------
-- R5. If the transfer switch is not pressed the system
-- shall not change the pilot flying side.
----------------------------------------------------------------------------------
guarantee "Pilot Flying Side Unchanged Unless Transfer Switch Pressed" :
  (Agree_Nodes.Duration(not Initializing and not PRESSED(TS)) > 25 =>
     (not (CHANGED(RPFS) or CHANGED(LPFS))));
```

Figure 34 – Pilot Flying Contract

Property R3 states that the pilot can always change the pilot flying side except when the system is in the process of switching sides. It makes use of three functions. *Duration(p)* has already been explained. *Since(p)* returns the number of steps since *p* was last true. *Within (p, n)* is true if *p* has been true within the last *n* steps. R3 states that if the Transfer Switch was pressed seven steps ago, and the system was not initializing and the Transfer Switch was not pressed in the 46 steps *previous* to that, then either the Right Side or the Left Side will become the pilot flying side within the last seven steps.

Attempting to prove this without further constraining the Transfer Switch generates a counterexample in which the press of the Transfer Switch is not seen by the not pilot flying side because its clock is false on the step when the Transfer Switch is pressed and the Transfer Switch returns to false before its clock ticks again. To fix this, we add the assumption *No Short Presses* at the start of the contract of Figure 34 which assumes that the Transfer Switch remains true for at least seven steps once it becomes true. Since the quasi-synchronous constraint requires that no clock can tick more than twice before every other clock has ticked at least once, a value of seven ensures that the clock of the not pilot flying side will tick at least once while the Transfer Switch is true, ensuring that its press is seen.

However, attempting to prove R3 even with this assumption reveals another counter example. This occurs when the next press of the Transfer Switch occurs so quickly that not pilot flying side fails to see the Transfer Switch go false. In this case, the last time the not pilot flying side observed the Transfer Switch, its value was true. The Transfer Switch subsequently became false, but the clock of the not pilot flying side did not tick until the after the Transfer Switch became true again, causing it to miss the press of the Transfer Switch. The fix for this situation is to add the constraint *No Rapid Presses* that assumes that the Transfer Switch remains false for at least seven steps once it becomes false.

With the addition of these two constraints, property R3 proves. Note that these assumptions are requirements pushed back onto the environment, specifically the implementation of the Transfer Switch, which must be satisfied for R3 to hold.

Property R4 states the system starts with the primary (left) side as the pilot flying side. The *followed by* operator -> defines a variable that has the value of its left hand side on the initial step and the value of its right hand side on all subsequent steps. So property R4 states that the output LPFS must be true on the first step and the output RPFS must be false on the first step. Following their initial value by true assures that R4 is trivially true for all subsequent steps.

Property R5 states that if the system has not been initializing and the Transfer Switch has not been pressed for twenty five steps, then the system will not change sides. In other words, that the system does not spontaneously change its state without an external stimuli.

Additional information needs to be added to some of the contracts of the AADL Pilot Flying model for the proofs of these properties to be completed. Just as the contract of Figure 34 is associated with the AADL system component for the Pilot Flying example, the contract shown in Figure 35 is associated with the AADL implementation of the Pilot Flying system. Since the AADL implementation specifies the subcomponents (Left Side, Right Side, LR_Bus, and RL_Bus) and their connections, this contract is able to refer to these subcomponents, something that could not be done in the contract of Figure 34.

The synchrony statement states that the clocks of the subcomponents are constrained to follow 2/1 quasi-synchrony[17], i.e., no subcomponent's clock can tick more than twice before every other subcomponent's clock has ticked at least once. Higher forms of quasi-synchrony, such as 3/2 quasi-synchrony (no clock can tick more than three times before every other subcomponent's clock has ticked at least twice) can be specified as "synchrony: 3, 2". It is also possible to specify the constraints between individual pairs of subcomponents A and B with a synchrony statement A,B:3,2, which states that nodes A, B must each observe 3/2 quasi-synchrony with respect to the other. For example, an alternative way to specify that the Pilot

---

[17] It is also possible to specify 2/1 quasi-synchrony as "synchrony: 2, 1", but this generates the less efficient set of clock constraints as discussed in Section 3.2.2.1.

Flying example observes 3/2 quasi-synchrony would be “synchrony **:** LS,LR:3,2 LS,RS:3,2 LS,RL:3,2 LR,RS:3,2 LR, RL:3,2 RS, RL:3,2;”.

```
synchrony : 2;

-- A subcomponent is initializing if its clock has not ticked once
eq LS_Initializing : bool = Agree_Nodes.ctF(1, LS._CLK);
eq RS_Initializing : bool = Agree_Nodes.ctF(1, RS._CLK);
eq LR_Initializing : bool = Agree_Nodes.ctF(1, LR._CLK);
eq RL_Initializing : bool = Agree_Nodes.ctF(1, RL._CLK);

-- The system is initializing if any component is not running
assert(Initializing = LS_Initializing or RS_Initializing or
                      LR_Initializing or RL_Initializing);

-- Set the outputs of each subcomponent during initialization
assert LS_Initializing => LS.PFS;
assert RS_Initializing => not RS.PFS;
assert LR_Initializing => LR.O;
assert RL_Initializing => not RL.O;

---------------------------------------------------------------------------------
-- This lemma speeds up higher level proofs
---------------------------------------------------------------------------------
lemma "Side Bus Consistency" :
       Agree_Nodes.Duration(LS.PFS and RS.PFS and RL.O and LR.O) <  10;
```

Figure 35 – Contract for the Pilot Flying Implementation

The next section specifies that each subcomponent is initializing until its clock has ticked at least once (the *ctF(n, clk)* function returns false until the clock *clk* has been true *n* times and returns true thereafter). The next statement assigns a definition to the *Initializing* variable introduced in the contract of Figure 34. This is possible since variables introduced in the contract of an AADL system are within scope of the contract of an implementation of that component. This allows us to introduce variables that can be used in specifying guarantees of a component, but whose definition requires knowledge of the component’s implementation.

The next section asserts that the outputs of the subcomponents have specific values during initialization. The Pilot Flying system is very sensitive to the output values of its subcomponents during initialization and only behaves correctly if these assertions hold. These are effectively design constraints that have to be implemented by the actual system.[18] Later, in the discussion of Active Standby example, we will consider a more robust design in which these constraints are relaxed.

The *Side Bus Consistency* lemma introduces an auxiliary property that can be proven true given the contracts of the system subcomponents. Since the Kind model checker proves properties through the use of *k*-induction, some properties may not prove without first proving auxiliary lemmas such as these that simplify the induction for the main property. This lemma

[18] For example, a component receiving a value from another component might default to the value specified in the assertion until the first actual value is received.

states that the longest the outputs of the four subcomponents can all be true is nine steps. Introduction of this lemma is necessary for the proofs of Figure 34 to complete. Auxiliary lemmas such as this are developed by studying the inductive counterexamples produced by the AGREE tool when a property does not prove. Even if a property can be proven without such auxiliary lemmas, the proofs will often complete faster if the simpler lemmas are proven first.

The time required to prove properties can also be reduced through the use of compositional verification, in which a contract for a component is proven about its implementation so that the simpler contract can be used in the verification of higher level properties. For example, the AGREE specification automatically generated from the SysML state machine for the Pilot Flying Side logic shown in Figure 27 can be proven to be equivalent to the specification shown in Figure 36.

```
eq ttF: bool = Agree_Nodes.tF(2);

guarantee "PFS Correct" : true ->  (PFS =
   if ttF then
      Get_Property(this, QS_Properties::Primary_Side)
   else if (pre(PFS) and riseOSPF) then
      false -- when the other side is observed becoming PFS
   else if (Agree_Nodes.Duration(not ttF and not(pre(PFS))) > 3 and riseTS) then
      true --  when TS is pressed while listening
   else pre(PFS));
```

Figure 36 – Contract for Pilot Flying Side Logic

Here, the function *tF(n)* returns a Boolean value that is true for the first *n* steps, then false for every step after that. It is introduced here since it takes one clock tick for a SysML state machine to enter its initial state. For example, in the quasi-synchronous Pilot Flying Side logic shown in Figure 17, the state machine enters its initial *Start* state at the end of the first step, and either the *Inhibited* or the *Pilot Flying* state at the end of its second step. In the contract of Figure 36, the value of PFS is unspecified in the initial step. In the next step, the state machine enters either the Pilot_Flying or Inhibited state and sets the value of PFS to the value of the AADL property *Primary_Side* for the component. For subsequent steps, the value of the PFS output is specified by one of the three else branches. This simpler contract is used by AGREE instead of the auto-generated contract of Figure 27.

Proving that implementation of Figure 27 satisfies the guarantee of Figure 36 requires extending the contract of Figure 27 with three auxiliary lemmas as shown in Figure 37. These lemmas are easily proven given the auto-generated contract of Figure 27. With them, the model-checker is able to prove the guarantee of Figure 36, allowing it to be used instead of the auto-generated contract of Figure 27.

```
lemma "Stop_Unreachable": (state != St_Stop);

lemma "Inhibit_Count_Bounded" :
    state = St_Inhibited => (inhibit_count >= 0 and inhibit_count <= 2);

lemma "PFS_State_Consistency" :
    (not (state = St_Start) => (PFS = (state = St_Pilot_Flying)));
```

Figure 37 – Auxiliary Lemmas for the Pilot Flying Side Logic Contract

With the auxiliary lemma for Side Bus Consistency and the use of compositional verification as just described, the proofs of system properties R1 through R5 can be completed using jKind on an Intel® Core™ i73720QM processor running at 2.6 GHz with 8 GB of RAM with the times shown in Table 3.

Table 3 – Pilot Flying Proof Times with AGREE and jKind

R1	R2	R3	R4	R5
276 sec	276 sec	344 sec	< 1 sec	166 sec

The time required to prove properties of quasi-synchronous systems is surprising, as is the number stimuli-free steps required to reach a quiescent state. To some extent, this is due to the inherent complexity of systems that are not synchronous. However, some of it is due to the fact that quasi-synchrony is a conservative over-approximation of what actually occurs in real systems. For example, in the Pilot Flying system it is often the case that only one of the four subcomponents can progress on a given step. Since the quasi-synchronous constraints allow the clocks of the other components to tick up to twice before the clock of that component is forced to tick, there are often sequences of six steps in counterexamples in which nothing occurs other than the ticking of the clocks of the other three components. This leads to very long traces that are expensive to verify using k-induction. This also manifests itself in the large number of steps required in reach a quiescent state.

4.1.2 Leader Selection Verification with AGREE and Kind.

Verification of the Leader Selection example described in Section 3.3.2 with AGREE is very similar to the verification of the Pilot Flying example. As indicated in Figure 18, the Leader Selection example has four clocks, one for each node and one for the Cross Node Bus. The contract for the Leader Selection system is shown in Figure 38.

```
eq Initializing: bool;

assume "Health1_Valid": Health1 >= 0 and  Health1 <= 100;
assume "Health2_Valid": Health2 >= 0 and  Health2 <= 100;
assume "Health3_Valid": Health3 >= 0 and  Health3 <= 100;

----------------------------------------------------------------------------------
-- CHANGED - returns true when input changes value
----------------------------------------------------------------------------------
node CHANGED (i : int) returns (r : bool);
  let r = false ->  not (i = pre(i)); tel;

----------------------------------------------------------------------------------
-- Defines of input changed
----------------------------------------------------------------------------------
eq input_changed: bool =
     CHANGED(Health1) or CHANGED(Health2) or CHANGED(Health3);

----------------------------------------------------------------------------------
-- R1. All nodes agree on the leader.
----------------------------------------------------------------------------------
guarantee "All Nodes Agree on the Leader" :
    Agree_Nodes.Duration(not Initializing and not input_changed) > 40 =>
         (Leader1 = Leader2) and (Leader2 = Leader3);

----------------------------------------------------------------------------------
-- R2. The leader is the healthiest node.
----------------------------------------------------------------------------------
guarantee "Leader is the Healthiest Node" :
    Agree_Nodes.Duration(not Initializing and not input_changed) > 40 =>
      ((Leader1 = 1 => (Health1 >= Health2 and Health1 >= Health3)) and
       (Leader1 = 2 => (Health2 >  Health1 and Health2 >= Health3)) and
       (Leader1 = 3 => (Health3 >  Health1 and Health3 >  Health2)) );

----------------------------------------------------------------------------------
-- R3. The leader shall not change unless a node's health changes.
----------------------------------------------------------------------------------
guarantee "Leader Unchanged Unless Health Changes" :
    Agree_Nodes.Duration(not Initializing and not input_changed) > 40 =>
         not (CHANGED(Leader1) or CHANGED(Leader2) or CHANGED(Leader3));
```

Figure 38 – Leader Selection Contract

To constrain the range of the input values for the health of each node, three assertions are added at the start of the formal requirements so that only input values of 0 to 100 are considered.

Unlike the Pilot Flying example that only had one Boolean input, Leader Selection inputs the three integer values for the health of each node. An external stimulus for the Leader Selection example thus occurs whenever any of these inputs change value. This is formally defined as the Boolean *input_changed*.

Property R1 states that if the system was not initializing and no input has changed for 40 steps, all three nodes will agree on who the leader is. Property R2 states that if the system is not initializing and no input has changed for 40 steps, the leader will be the healthiest node (with ties awarded to the node with the smaller index). Property R3 states that if the system is not initializing and no input has changed for 40 steps, the leader will not change.

As in the Pilot Flying example, compositional verification using simpler contracts for each Node and the Cross Node Bus are used to reduce the time needed for verification of these three requirements. The time needed to complete the proofs on an Intel® Core™ i73720QM processor running at 2.6 GHz with 8 GB of RAM using the jKind model checker are shown in Table 4.

Table 4 – Leader Selection Proof Times with AGREE and jKind

R1	R2	R3
664 sec	664 sec	667 sec

4.1.3 Active Standby Verification with AGREE and Kind.

The Active Standby example described in Section 3.3.3 is a more robust version of the Pilot Flying example suitable for use in a highly critical system such as the Primary Flight Control System. The key differences between it and the Pilot Flying example are that a side can fail at any time and later heal and that the active side can be changed by a side failing, by one side becoming more available than the other side, or by the pilot manually changing the side. The top level contract for the Active Standby system is shown in Figure 39 through Figure 41.

```
eq NeitherSideFailed : bool = not (Side1Failed or Side2Failed);

eq Side1FullyAvail : bool = Side1SubsystemStatus.Subsystem_A_Avail and
                            Side1SubsystemStatus.Subsystem_N_Avail;

eq Side2FullyAvail : bool = Side2SubsystemStatus.Subsystem_A_Avail and
                            Side2SubsystemStatus.Subsystem_N_Avail;

eq Side1AvailabilityChanged : bool = Agree_Nodes.Changed(Side1FullyAvail);

eq Side2AvailabilityChanged : bool = Agree_Nodes.Changed(Side2FullyAvail);

eq NoChangeInAvailability : bool =
     not (Side1AvailabilityChanged or Side2AvailabilityChanged);

eq NoChangeInFailedStatus : bool = not (Agree_Nodes.Changed(Side1Failed) or
                                        Agree_Nodes.Changed(Side2Failed));

eq ManualSelectionPressed : bool = Agree_Nodes.Rise(ManualSelection);

eq Side1_Initializing: bool;
eq Side2_Initializing: bool;

eq Initializing : bool;

-- Each side is failed during initialization
assume "Side1 Failed Until Initialized" : Side1_Initializing => Side1Failed;
assume "Side2 Failed Until Initialized" : Side2_Initializing => Side2Failed;

-- At most one side can be failed after initialization
assume "At Most One Side Failed": not(Side1Failed and Side2Failed);

-- Manual selection presses must be long enough to be seen by both sides
assume "No Short Presses": Agree_Nodes.True_At_Least(ManualSelection, 7);
assume "No Rapid Presses": Agree_Nodes.True_At_Least(not ManualSelection, 7);

-- A side cannot heal too quickly after failing
assume "No Fast Healing":  Agree_Nodes.True_At_Least(Side1Failed, 7) and
                           Agree_Nodes.True_At_Least(Side2Failed, 7);
```

Figure 39 – Active Standby Contract (Part 1)

The contract first specifies definitions helpful in writing the system requirements. This is followed by assumptions the system must make about its inputs in order for the verification to succeed. For example, we assume that during the initialization of each side, it behaves as though it is failed. This assumption must be satisfied by the implementation of each side. We also assume that once the system is initialized, at most one side can be failed at a time. This assumption must be validated as part of the system safety assessment. Just as for the Pilot Flying example, we assume button presses of the *Manual Selection* switch cannot be so short or so rapid that they are not seen by a component. New to the Active Standby system is the assumption that a side cannot heal too rapidly after failing.

The system assumptions are followed by the system requirements shown in Figure 40.

```
---------------------------------------------------------------------------------
-- R0. At least one side should always be active.
--------------------------------------------------------------------------------
guarantee "At Least One Side Active":
   Agree_Nodes.Duration(not Initializing and
         not (Side1Status.Active or Side2Status.Active)) < 25;

--------------------------------------------------------------------------------
-- R1. Both sides should agree on which side is active provided neither side has
-- failed, the availability of a side has not just changed and the pilot has not
-- just made a manual selection.
--------------------------------------------------------------------------------
guarantee "Both Sides Agree" :
   (Agree_Nodes.Duration(not Initializing and NeitherSideFailed and
          NoChangeInAvailability and not ManualSelectionPressed) > 43 =>
   (Side1Status.Active = not Side2Status.Active));

---------------------------------------------------------------------------------
-- R2. A side that is not fully available should not be the active side if
-- the other side is fully available, provided neither side has failed, the
-- availability of a side has not changed, and the pilot has not made a manual
-- selection.
-------------------------------------------------------------------------------
guarantee "Side 1 Not Fully Available" :
   (Agree_Nodes.Duration(not Initializing and NeitherSideFailed and
          NoChangeInAvailability and not ManualSelectionPressed) > 31 =>
   ((Side2FullyAvail and not Side1FullyAvail) => Side2Status.Active));

guarantee "Side 2 Not Fully Available" :
   (Agree_Nodes.Duration(not Initializing and NeitherSideFailed and
          NoChangeInAvailability and not ManualSelectionPressed) > 31 =>
   ((Side1FullyAvail and not Side2FullyAvail) => Side1Status.Active));
```

Figure 40 – Active Standby Contract (Part 2)

Requirement R0 states that the longest neither side is the active side after initialization is 25 steps or less. Ideally, there would always be an active side, but since the active side can fail at any time and it will take a finite time for the other side to become the active side, this is the best that can be done assuming quasi-synchronous clocks.

Property R1 states that the two sides will agree on which side is the active side once a quiescent state is reached. Stimuli can be provided to the Active Side logic whenever the pilot presses the *Manual Selection* button, the availability of the aircraft systems on a side changes, or a side hosting the Active Standby logic fails. Property R2 states that once a quiescent state is reached, a side should not be the active side unless that side's aircraft systems are at least as available as those on the other side. Requirements R3 through R5 are shown in Figure 41.

```
---------------------------------------------------------------------------------
-- R3. The pilot can always change the active side except if a side is failed,
-- the availability of a side has changed, or the pilot has requested a manual
-- selection
---------------------------------------------------------------------------------
guarantee "Pilot Can Change Active Side":
   Agree_Nodes.Since(
     Agree_Nodes.Duration(pre(not Initializing and NeitherSideFailed and
           NoChangeInAvailability and not ManualSelectionPressed)) > 43 and
           ManualSelectionPressed) = 7 =>
   Agree_Nodes.Within(Agree_Nodes.Rise(Side1Status.Active) or
                      Agree_Nodes.Rise(Side2Status.Active),7);

---------------------------------------------------------------------------------
-- R4. If a side is failed then the other side should become the active side
-- unless the other side is also failed.
---------------------------------------------------------------------------------
guarantee "Side1 Failed" :
   (Agree_Nodes.Duration(not Initializing and Side1Failed and not Side2Failed) >
      25 => Side2Status.Active);

guarantee "Side2 Failed" :
   (Agree_Nodes.Duration(not Initializing and Side2Failed and not Side1Failed) >
      25 => Side1Status.Active);

---------------------------------------------------------------------------------
-- R5 The active side should not change unless the availability of a side changes,
-- the failed status of a side changes, or manual selection is selected by the
-- pilot.
---------------------------------------------------------------------------------
guarantee "No Spontaneous Change" :
    (Agree_Nodes.Duration(not Initializing and NoChangeInAvailability and
           NoChangeInFailedStatus and not ManualSelectionPressed) > 44 =>
    (not (Agree_Nodes.Changed(Side1Status.Active) or
          Agree_Nodes.Changed(Side2Status.Active))));
```

Figure 41 – Active Standby Contract (Part 3)

Property R3 states that the pilot can always change the active side if the system is in a quiescent state (otherwise it is already in the process of changing sides). However, the component clocks may delay the response to a press of the *Manual Selection* button, the formalization of this property is rather complicated. The *Since(p)* function returns the number of steps since *p* has been true and *Within(p ,n)* returns true if *p* has been true during the preceding *n* steps. Property R3 asserts that if it has been seven steps since the *Manual Selection* button was pressed while the system was in a quiescent state[19] on the previous step, then either Side1 or

[19] Duration(not Initializing and NeitherSideFailed and NoChangeInAvailability and not ManualSelectionPressed)) > 43.

Side2 will become the active side within the next seven steps. Property R4 states that if a side fails, then the other side should become the active side within 25 steps unless the other side is also failed. Finally, property R5 states that the system does not spontaneously change the active side unless the pilot presses the *Manual Selection* button, the availability of the aircraft systems on a side changes, or a side fails.

The Active Standby system makes some use of compositional verification to speed up the proofs, but constructing a more abstract contract for the subsystem illustrated in Figure 21 and Figure 22 is a non-trivial task. Instead, the contract for the implementation shown in Figure 21 uses the "lift" operator to tell AGREE to construct a contract based on the contracts of its subcomponents as shown in Figure 42. This eliminates the need to manually construct a contract, but it also eliminates the opportunity to reduce proof time by substituting a simpler contract.

```
        lift Monitor;
        lift ActiveSideLogic;
        lift OtherSideDemux;
        lift ThisSideMux;
        lift riseOSA;
        lift ThisSubDemux;
        lift OthrSubDemux;
        lift riseMS;
```

Figure 42 – Constructing a Contract Using Lift

The contract for the top-level *implementation* of the Active Side system is shown in Figure 43. It includes the usual synchrony command and the definition of when each component is initializing. Note that the entire system is considered to be initializing for an additional seven steps after each component finishes initializing. This is necessary to avoid a logical inconsistency due to the assumptions that each component is failed while it is initializing, a component cannot heal (become not failed) in fewer than seven steps, and that only one side can be failed once the overall system is initialized.   In effect, the system has to be considered to be initializing for seven steps after all its components have initialized to provide one side sufficient time to heal without violating the assumption that a side must stay failed for at least seven steps after failing.

```
synchrony : 2;

-- Each side is initializing until its clock has ticked twice
assert Side1_Initializing = Agree_Nodes.ctF(1,Side1._CLK);
assert Side2_Initializing = Agree_Nodes.ctF(1,Side2._CLK);

-- Each bus is initializing until its clock has ticked twice
eq Bus_LR_Initializing : bool = Agree_Nodes.ctF(1,Bus12._CLK);
eq Bus_RL_Initializing : bool = Agree_Nodes.ctF(1,Bus21._CLK);

-- The system remains in initialization for seven steps after all its components
-- have initialized to ensure that at least one side has time to heal
-- (both sides are assumed failed during the side's initialization)
assert(Initializing =
   (Agree_Nodes.Since(Side1_Initializing or Bus_LR_Initializing or
                       Side2_Initializing or Bus_RL_Initializing) < 8));

-- This lemma speeds up the system level proofs
lemma "Side Bus Consistency" :
   (Agree_Nodes.Duration(not Initializing and
                         Side1.ThisSideStatus.Active and
                          Side2.ThisSideStatus.Active and
                           Bus12.O.Active and Bus21.O.Active) < 12);
```

Figure 43 – Contract for Active Side Implementation

Given the assumptions of Figure 39, properties R0 through R5 can be proved on an Intel® Core™ i73720QM processor running at 2.6 GHz with 8 GB of RAM using the jKind model checker with the times shown in Table 5.

Table 5 – Active Standby Proof Times with AGREE and jKind

R0	R1	R2	R3	R4	R5
1752 sec	3215 sec	1088 sec	3215 sec	608 sec	1890 sec

4.1.4 WBS Verification with AGREE and Kind.

The top level contract for the Wheel Braking System described in Section 3.3.4 is shown in Figure 44 and Figure 45. As discussed in section 3.3.4, the WBS model is derived from the accident report and includes a number of assumptions and may not accurately describe the actual implemented system. For this reason, the verification described here only claims to identify errors in this model. The contract begins with definitions helpful in specifying properties and assumptions about the inputs as shown in Figure 44.

```
eq Initializing : bool;

eq COM1Failed: bool = Agree_Nodes.Latch(FailCOM1);
eq COM2Failed: bool = Agree_Nodes.Latch(FailCOM2);
eq MON1Failed: bool = Agree_Nodes.Latch(FailMON1);
eq MON2Failed: bool = Agree_Nodes.Latch(FailMON2);

eq No_Failed_Component: bool =
   not (COM1Failed or COM2Failed or MON1Failed or MON2Failed);

eq At_Most_One_Failed_Component: bool =
   (COM1Failed => not (COM2Failed or MON1Failed or MON2Failed)) and
   (COM2Failed => not (COM1Failed or MON1Failed or MON2Failed)) and
   (MON1Failed => not (COM1Failed or COM2Failed or MON2Failed)) and
   (MON2Failed => not (COM1Failed or COM2Failed or MON1Failed));

assume "At Most One Failed Component" : At_Most_One_Failed_Component;

assume "Fixed Pedal Pressure" : Pedal = 100;

assume "Only One Button Pressed at a Time" :
        (Panel.LO  => not (Panel.MED or Panel.MAX)) and
        (Panel.MED => not (Panel.LO  or Panel.MAX)) and
        (Panel.MAX => not (Panel.LO  or Panel.MED));
```

Figure 44 – WBS Contract (Part 1)

The variables COM1Failed, COM2Failed, MON1Failed, and MON2Failed latch the inputs indicating if the associated component has failed. For the WBS system, we will treat a component as permanently failed if it has failed at any time. The variables No_Failed_Component and At_Most_One_Failed_Component allow us to state properties about the cases where no component has failed and a single component has failed. Based on the system safety analysis we can assume that at most one component will ever fail during a normal period of operation. This is captured in the assertion “At Most One Failed Component”.

The assumption “ Fixed Pedal Pressure” is made to eliminate counterexamples caused by rapid changes in the pedal pressure. Since the analysis of the WBS system focuses on how pressing the LO, MED, or MAX buttons can cause the logic error identified in the accident report, the portion of the model dealing with pedal pressure and anti-skid computation have been deliberately simplified. As a consequence, rapid changes in the pedal pressure can cause spurious counterexamples. A higher fidelity model would include limits on how quickly the pedal pressure could change and thresholds for the detection of disagreements leading to error states.

The last assumption states that only one button is ever pressed at a time. This is essentially a design constraint on how the button panel is implemented. The guarantees for the WBS are shown in Figure 45.

```
---------------------------------------------------------------------------------
-- R1. At least one channel shall be error free if no components have failed.
---------------------------------------------------------------------------------
guarantee "At Least One Channel Error Free - No Failures" :
   No_Failed_Component =>
      (not Initializing => (not CH1.Error or not CH2.Error));

---------------------------------------------------------------------------------
-- R2. At least one channel shall be error free
-- if at most one component has failed.
---------------------------------------------------------------------------------
guarantee "At Least One Channel Error Free - One Failure" :
   At_Most_One_Failed_Component =>
      (not Initializing => (not CH1.Error or not CH2.Error));

---------------------------------------------------------------------------------
-- R3. At least one channel shall be active if no components have failed.
---------------------------------------------------------------------------------
guarantee "At Least One Channel Active - No Failures" :
   No_Failed_Component =>
       (not Initializing => CH1.Active or CH2.Active);

---------------------------------------------------------------------------------
-- R4. At least one channel shall be active if at most one component has failed.
---------------------------------------------------------------------------------
guarantee "At Least One Channel Active - One Failure" :
   At_Most_One_Failed_Component =>
     Agree_Nodes.Duration(not Initializing and not(CH1.Active or CH2.Active)) < 7;

---------------------------------------------------------------------------------
-- R5. At most one channel shall be active if no components have failed.
---------------------------------------------------------------------------------
guarantee "At Most One Channel Active - No Failures" :
   No_Failed_Component =>
     Agree_Nodes.Duration(not Initializing and CH1.Active and CH2.Active) < 7;

---------------------------------------------------------------------------------
-- R6. At most one channel shall be active if at most one component has failed.
---------------------------------------------------------------------------------
guarantee "At Most One Channel Active - One Failure" :
   At_Most_One_Failed_Component =>
     Agree_Nodes.Duration(not Initializing and CH1.Active and CH2.Active) < 7;
```

Figure 45 – WBS Contract (Part 2)

The first two guarantees state that at least one WBS channel will be error free if no component and at most one component is failed. Of course, the proof for when no component is failed is redundant since it is implied by the proof for when at most one component is failed. Also, the antecedent *At_Most_One_Failed_Component* is unnecessary since we assume in

Figure 44 that it is always the case that at most one component is failed. It is included in specification of the second property both as documentation and to ensure the property is still stated correctly even if the assumption were removed. The next two guarantees state that at least one channel is active if no component or at most one component is failed. Finally, the last two guarantees state that at most one channel is active if no component or at most one component is failed.

The COM and MON units compute their next brake mode (corresponding to Figure 24) using the Brake_Mode function defined in Figure 46.

```
node Brake_Mode(mode: int, lo: bool, med: bool, max: bool) returns(r: int);
let
  r = Agree_Constants.MANUAL ->
      if (pre(mode) = Agree_Constants.LO        and Rise(lo))  then
         Agree_Constants.MANUAL
      else if (pre(mode) = Agree_Constants.MED  and Rise(med)) then
         Agree_Constants.MANUAL
      else if (pre(mode) = Agree_Constants.MAX  and Rise(max)) then
         Agree_Constants.MANUAL
      else if (pre(mode) != Agree_Constants.LO  and Rise(lo))  then
         Agree_Constants.LO
      else if (pre(mode) != Agree_Constants.MED and Rise(med)) then
         Agree_Constants.MED
      else if (pre(mode) != Agree_Constants.MAX and Rise(max)) then
         Agree_Constants.MAX
      else pre(mode);
tel;
```

Figure 46 – Computation of the WBS Brake Mode

The COM and MON units compute their break command using the Brake_Cmd function defined in Figure 47. The actual values returned are irrelevant to the analysis so long as they are different.

```
node Brake_Cmd(mode: int, pedal: int) returns(r: int);
let
    r = if (mode = Agree_Constants.LO)  then 170 else
        if (mode = Agree_Constants.MED) then 340 else
        if (mode = Agree_Constants.MAX) then 510 else pedal;
tel;
```

Figure 47 – Computation of the WBS Brake Command

The contract for the COM units is shown in Figure 48. Note that the brake command is only specified if the COM unit is not failed. Otherwise, the brake command is unspecified and can take on any value.

```
eq Mode: int = Agree_Nodes.Brake_Mode(Mode, Panel.LO, Panel.MED, Panel.MAX);

-- The CMD output is reliably output if the component is not failed
guarantee "Valve Command" :
   not Fail => CMD = Agree_Nodes.Brake_Cmd(Mode, Pedal);
```

Figure 48 – Contract for WBS COM Unit

The contract for the MON units is shown in Figure 49. The Error variable defines whether the channel is in error. This occurs when the brake command computed by the COM unit differs from the brake command computed by the MON unit for more than six steps. Later, we prove that the longest the two values can differ due to the ticking of the clocks is six steps, so a miscompare of more than six steps is sufficient to identify a channel error.

```
eq Primary_Side: bool = Get_Property(this, QS_Properties::Primary_Side);

eq Mode: int = Agree_Nodes.Brake_Mode(Mode, Panel.LO, Panel.MED, Panel.MAX);

eq Error: bool =
   Agree_Nodes.Latch(Agree_Nodes.Duration(not (Status.Cmd = CMD_From_COM)) >= 7);

eq Active : bool =
   -- first side started becomes the active side
    (not Sync_From.Active ->
   -- A side in error is never active
        if (Error) then false
        -- Resolve ties at start-up in favor of the primary side
        else if (pre(Active) and Sync_From.Active and
                 not Sync_From.Error and not Primary_Side) then false
        -- Other side is no longer active or is in error so we become active
        else if (not pre(Active) and
                (not Sync_From.Active or Sync_From.Error)) then true
        -- Otherwise no change
        else pre(Active));

-- The CMD output is reliably output if the component is not failed
   guarantee "MON Brake Command" :
        not Fail => (Status.Cmd = Agree_Nodes.Brake_Cmd(Mode, Pedal));

-- The Error status is reliably output even if the component is failed
   guarantee "Status Error" : Status.Error = Error;
   guarantee "Sync To Error" : Sync_To.Error = Error;

-- The Active status is reliably output even if the component is failed
   guarantee "Status Active" : Status.Active = Active;
   guarantee "Sync To Active" : Sync_To.Active = Active;
```

Figure 49 – Contract for WBS MON Unit

The Active variable holds whether the channel believes it is the active side. Active status is initially given to the first MON unit to execute, with ties broken in favor of the Primary Side on the next step. A side will become active if it sees the other side declare itself in error or no longer active.

Just as with the COM unit, a MON unit will only reliably output it's computed brake command if it is not failed. However, a MON unit must reliably output its Error and Active status if the two channels are to agree on who is the active side. If this is not an implementable guarantee, the WBS architecture would have to be revised.

In addition, additional definitions, implementation constraints, and auxiliary lemmas need to be added to the contract for the BSCU implementation just as in the other examples. These include the lemmas shown in Figure 50 stating that a channel can only be declared in error if it's COM or MON component has failed.

```
lemma "CH1 Error Only if Failed" : CH1.Error => COM1Failed or MON1Failed;
lemma "Ch2 Error Only if Failed" : CH2.Error => COM2Failed or MON2Failed;
```

Figure 50 – Contract for WBS MON Unit

Trying to prove the lemmas of Figure 50 generates the counterexample[20] of Figure 51.

Step	0	1	2	3	4	5	6	7	8	9	10	11
BSCU_BSCU_Impl_Instance												
CH1.Error	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE
CLOCK_COM1	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	FALSE	FALSE	TRUE	TRUE	TRUE	TRUE
CLOCK_MON1	FALSE	FALSE	TRUE	TRUE	FALSE	FALSE	FALSE	TRUE	TRUE	TRUE	TRUE	TRUE
Panel.LO	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE
Panel.MAX	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
Panel.MED	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
COM1												
COM1.CMD	101	101	101	101	100	100	100	100	100	100	100	100
MON1												
MON1.CMD_From_COM	101	101	101	101	100	100	100	100	100	100	100	100
MON1.Status.Cmd	100	100	100	100	100	100	100	170	170	170	170	170
MON1.Status.Error	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE

Figure 51 – Counterexample Caused by Short Button Presses

The counterexample of Figure 51 demonstrates how a channel can be declared in error even when neither it's COM nor MON unit have failed. In step 6, COM1 and MON1 are in MANUAL mode and are generating brake command values of 100 (the pedal pressure). In step 7, the LO button on the BSCU panel is pressed for one step. The clock for MON1 is true on step 7, so it observes the button press and switches into LO mode and outputs 170 as its brake command. However, the clock for COM1 is false on step 7, so it does not observe the button press and remains in MANUAL mode. At this point, the actual fault has occurred, but the error is not reported until MON1 observes a miscompare for seven steps in step 11. While it may appear that step 11 is too soon for a miscompare of seven steps to occur, note that clock of MON1 is true in step 2 and step 3 and in steps 7 through 11. Since the brake command from COM1 differs from the value computed by MON1 on all those steps, to MON1 it appears that a miscompare has occurred for seven steps, causing it to declare itself in error.

[20] To generate this counterexample it is necessary to add an assumption that no button presses occur in the first six steps. This eliminates a spurious counterexample that arises in the absence of the full set of assumptions. Since it is possible that no button presses occur in the first six steps, the resulting counterexample is still valid.

The cause of this error is that the button press is so short that it was observed by MON1 but missed by COM1 since its clock was false. The same scenario can occur in channel 2. Thus, even though none of the components had failed, both channels can declare themselves in error. This is the error described in the accident report. One fix to this problem is to force button presses to last long enough that they will always be observed by all components. This is easily done in the model by adding the assumptions shown in Figure 47. The first assumption requires that a button press must stay true for at least seven steps, ensuring that all nodes will see the button press. The second assumption requires that button presses stay true for at most 20 steps. This eliminates counterexamples that occur when a button is never released (any large value could be used).

```
assume "Button Presses are Long Enough to be Seen by All Nodes" :
       Agree_Nodes.True_At_Least(Panel.LO,  7) and
       Agree_Nodes.True_At_Least(Panel.MED, 7) and
       Agree_Nodes.True_At_Least(Panel.MAX, 7);

assume "Button Presses are Bounded in Duration" :
       Agree_Nodes.True_At_Most(Panel.LO,  20) and
       Agree_Nodes.True_At_Most(Panel.MED, 20) and
       Agree_Nodes.True_At_Most(Panel.MAX, 20);
```

Figure 52 – Constraining the Duration of WBS Button Presses

However, trying to verify the WBS system with these assumptions generates another counterexample[21] shown Figure 53.

Step	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
BSCU_BSCU_Impl_Instance																	
CH1.Error	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE
CH2.Error	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
CLOCK_COM1	TRUE	TRUE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE	FALSE
CLOCK_MON1	TRUE	TRUE	TRUE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE
Panel.LO	TRUE	TRUE	FALSE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
Panel.MAX	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
Panel.MED	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
COM1																	
COM1.CMD	170	170	170	170	170	170	170	170	170	170	170	170	170	170	170	170	170
MON1																	
MON1.CMD_From_COM	170	170	170	170	170	170	170	170	170	170	170	170	170	170	170	170	170
MON1.Status.Cmd	170	170	170	170	100	100	100	100	100	100	100	100	100	100	100	100	100
MON1.Status.Error	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE

Figure 53 – Counterexample Caused by Rapid Button Presses

In step 13, both the COM1 and MON1 units are in LOW mode and are generating brake command values of 170. The LO button on the BSCU panel is true on step 13, but neither unit responds to it since it was also true on the preceding six steps. In step 14, the LO button becomes false and then becomes true again on step 15. The MON1 unit does not respond to the LO button press on step 15 because its clock is false on that step. However, on step 16 its clock is true. The

[21] To generate the counterexample in reasonable time, we set the quasi-synchronous constraint to 1/1 quasi-synchrony, i.e., no clock can tick more than once before every other clock has ticked once. This constrains the clocks to tick in the same order, but not necessarily at the same time. Since a 2/1 quasi-synchronous system can always behave like a 1/1 quasi-synchronous system, a counterexample produced assuming 1/1 quasi-synchrony is still a valid counterexample for the 2/1 quasi-synchronous system.

last time MON1 saw the LO button on step 14, the LO button was false, so on step 16 the MON1 unit sees the LO button press and reverts from LO mode to MANUAL mode, outputting a brake command of 100 (the pedal pressure). In contrast, the clock of COM1 was false on step 14, so it never saw the LO button go false. On step 15, it observes that the LO button is true, but the last time it observed the LO button on step 13, it was also LO, so it fails to observe the LO button press and remains in LO mode. At this point, the actual fault has occurred, but the error is not reported until MON1 observes a miscompare for seven steps in step 28 and declares itself in error.

In contrast to the counterexample of Figure 51 which was caused by the button press being too short, this counterexample was caused by the button press being false for such a short time that the next button press was missed by COM1. In effect, the next button press arrived too quickly. One fix for this problem is to force button presses to remain false long enough to ensure that the next button press is observed by all components, i.e., to ensure that button presses cannot arrive too quickly. This is easily done in the model by adding the assumptions shown in Figure 54 to ensure that a button must remain false for at least seven steps.

```
assume "Button Presses Do Not Occur Too Quickly" :
        Agree_Nodes.True_At_Least(not Panel.LO,  7) and
        Agree_Nodes.True_At_Least(not Panel.MED, 7) and
        Agree_Nodes.True_At_Least(not Panel.MAX, 7);
```

Figure 54 – Constraining the Arrival of WBS Button Presses

As it turns out, the proofs still do not complete even with these assumptions. Examining the inductive counterexample[22] indicates that the problem lies with button presses that move a COM or a MON unit from one mode to a different mode to a third mode every seven steps. Gradually increasing the number of steps a button must remain true or false reveals that all requirements prove when the button presses are assumed to be true for at least 13 steps and false for at least 13 steps. The final set of assumptions that must be made about the button presses to complete the proofs of Figure 45 are shown in Figure 55.

With these assumptions, properties R1 through R6 can be proved on an Intel® Core™ i73720QM processor running at 2.6 GHz with 8 GB of RAM using the jKind model checker with the times shown in Table 6.

Table 6 – WBS Proof Times with AGREE and jKind

R1	R2	R3	R4	R5	R6
816 sec	816 sec	816 sec	816 sec	34 sec	34 sec

[22] An inductive counterexample is generated when a proof fails to complete or generate a true counterexample before the timeout is reached. Examining the inductive counterexample can provide clues as to why the proof is not completing.

```
assume "Only One Button Pressed at a Time" :
       (Panel.LO  => not (Panel.MED or Panel.MAX)) and
       (Panel.MED => not (Panel.LO  or Panel.MAX)) and
       (Panel.MAX => not (Panel.LO  or Panel.MED));

assume "Button Presses are Long Enough to be Seen by All Nodes" :
        Agree_Nodes.True_At_Least(Panel.LO,  13) and
        Agree_Nodes.True_At_Least(Panel.MED, 13) and
        Agree_Nodes.True_At_Least(Panel.MAX, 13);

assume "Button Presses are Bounded in Duration" :
        Agree_Nodes.True_At_Most(Panel.LO,  20) and
        Agree_Nodes.True_At_Most(Panel.MED, 20) and
        Agree_Nodes.True_At_Most(Panel.MAX, 20);

assume "Button Presses Do Not Occur Too Quickly" :
       Agree_Nodes.True_At_Least(not Panel.LO,  13) and
       Agree_Nodes.True_At_Least(not Panel.MED, 13) and
       Agree_Nodes.True_At_Least(not Panel.MAX, 13);
```

Figure 55 – WBS Button Press Assumptions

4.2 Formal Verification with Uppaal

This section discusses issues in formally verifying the example problems with Uppaal. As discussed in Section 3.2.2.2, in Phase 1 of the project, verification with Uppaal was done with actual values for period and jitter assigned to each clock. This is not as general as verifying all possible quasi-synchronous assignments of period and jitter as was done with Kind, but it has the advantage that it produces actual values for how long it takes the system to reach a quiescent state after being stimulated. In Phase II of the project, the verification was done constraining the clocks to satisfy the quasi-synchronous constraint without using real-time values (also discussed in Section 3.2.2.2). Verification of the Pilot Flying example with Uppaal is discussed in Section 4.2.1 and verification of the Leader Selection example is discussed in Section 4.2.2. The Uppaal model checker timed out without completing verification of the properties for the Active Standby and WBS, so results for those examples are not presented.

4.2.1 Pilot Flying Verification with Uppaal.

This section discusses the formal verification of the Pilot Flying example with Uppaal. Section 4.2.1.1 discusses verification using real-time clocks and Section 4.2.1.2 discusses verification using quasi-synchronous clocks.

4.2.1.1 Pilot Flying Verification in Uppaal with Real-time Clocks

The values chosen for the period and jitter of each clock in the Pilot Flying example are shown in Table 7 (the units in which time are measured are immaterial).These are easily seen to satisfy the quasi-synchronous constraints described in Section 3.2.1.

Table 7 – Pilot Flying Clock Periods and Jitter for Real-time Uppaal Verification

Clock	Subsystem	Period	Jitter
CLK1	Left Side	46	4
CLK2	LR Bus	50	0
CLK3	Right Side	54	2
CLK4	RL Bus	78	5

To formally verify the Pilot Flying example in Uppaal, the auto-generated model must be supplemented with a process to generate the system inputs. Inputs in Uppaal are typically modeled as events arriving from the environment. However, most avionics systems are sampled data systems that read all the input values at the start of each step. In the Pilot Flying example, there is a single input, the Transfer Switch that each side reads through its *TS* port as shown in Figure 32.

In Uppaal this is modeled as a global variable *TS* that is passed to both sides.[23] To emulate this as a sampled input in Uppaal, we add the process shown in Figure 56. This process has a single location *Start* with a single transition. In making this transition, it randomly sets the value of its local Boolean variable *next_TS*, calls the local function *IsPressed* with argument

[23] Since only the Pilot Not Flying side listens to this port and since there is never more than one Pilot Not Flying side, we do not need to model asynchronous arrivals of this input at each side.

next_TS, then sets the new value of *TS* to *next_TS*. The function *IsPressed* has two side-effects. It sets the local variable *pressed* to *(not TS and next_TS)* and, if *pressed* is true, it resets a local clock *q* to 0. In this way, Uppaal treats *TS* as sampled input that changes randomly. The local variable *pressed* detects a rising edge of *TS* indicating whether the Transfer Switch was pressed. The local clock *q* tracks the time since *TS* was last pressed. *Pressed* and *q* are used only in stating properties to be checked and are not used in the model itself.

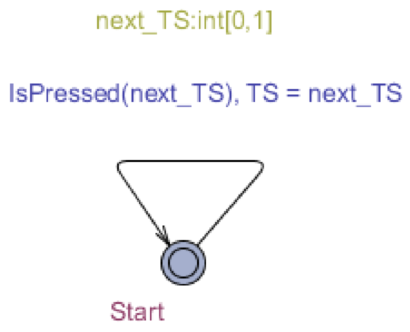


Figure 56 – Uppaal Input Process for Pilot Flying Example

With a process defined to generate inputs, the next step is to formally state the Pilot Flying requirements. The first Pilot Flying requirement R1 from Section 3.3 states that at least one side is always the pilot flying side. The Uppaal query for this is shown in Figure 57. This property states that for all states and all paths (i.e., always henceforth) either the output *LPFS* is true or the output *RPFS* is true. Uppaal is able to prove this in 142 seconds.

```
A[] LPFS || RPFS
```

Figure 57 – Pilot Flying Requirement R1 for Real-time Uppaal

Formally stating requirement R2 that both sides shall agree on the pilot flying side is slightly more complicated. Just as in Section 4.1.1, property R2 only holds when the system is not switching sides, i.e., when in a quiescent state. This can be stated formally in Uppaal as shown in Figure 58. Recall that *q* is the time since the Transfer Switch was last pressed. The property of Figure 58 thus states that for all states and all paths, if the Transfer Switch has not been pressed for 240 time units[24], the output *LPFS* will not equal output *RPFS*, i.e., the two sides will agree on who is the pilot flying side. This property can be proven with Uppaal in approximately 191 seconds.

```
A[] q >= 240  imply LPFS != RPFS
```

Figure 58 – Pilot Flying Requirement R2 for Real-time Uppaal

Property R3 stating that pressing the Transfer Switch will change the pilot flying side presents a slightly different challenge. Recall that in verifying this property using Kind, we stated that if in the previous step the Transfer Switch had not been pressed for 46 steps and the side was *not* the pilot flying side, pressing the Transfer Switch so that the not pilot flying side observed it

[24] This property fails for values smaller than 240.

being pressed would cause the side to become the pilot flying side. Uppaal does provide an *eventually* temporal operator, but it does not provide a *next state* temporal operator. However, a similar capability can be achieved by defining *synchronous observer* processes such as that shown in Figure 59.

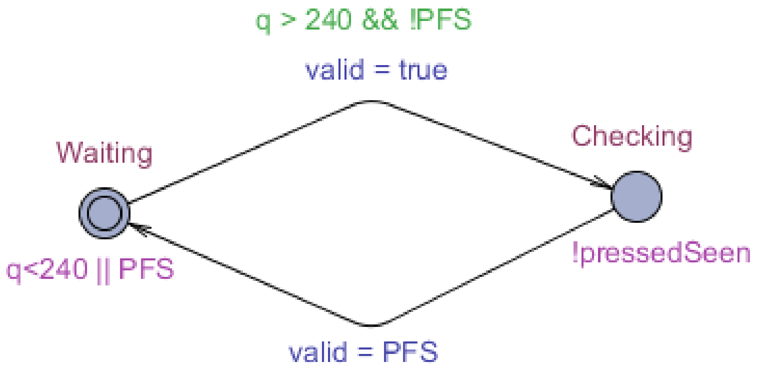


Figure 59 – Observer for Pilot Flying Requirement R3 in Real-time Uppaal

A synchronous observer executes in parallel with the system and emits the value true when the property holds and the value false when the property is falsified. The observer of Figure 59 transitions from location *Waiting* to location *Checking* whenever the Transfer Switch has not been pressed for more than 240 time units and the side is not the pilot flying side, setting *valid* to true since the property has not yet been falsified. If the Transfer Switch is not pressed while in location *Waiting*, no change occurs since property R3 is only concerned with what happens when the Transfer Switch is pressed. If the Transfer Switch is pressed and *seen* by the side while in location *Checking*, it returns to location *Waiting*, setting valid to true if the side is now the pilot flying side (PFS is true) and to false if the side is not the pilot flying side (PFS is false). For the pilot flying system, the template for Figure 59 is named R3 and takes parameters *&PFS* and *&pressedSeen*. Process instances for the left and right sides are instantiated as shown in Figure 60.

```
R3_left  = R3(LPFS, LS_riseTS_O)
R3_right = R3(RPFS, RS_riseTS_O)
```

Figure 60 – Uppaal Instantiations of Pilot Flying Property R3

The variables *LS_riseTS_O* and *RS_riseTS_O* contain the values of the output of the *riseTS* component (Figure 32) indicating if the press of the Transfer Switch was actually seen by the side. The query presented to Uppaal is shown in Figure 61. Unfortunately, the proof of property R3 does not complete. Instead, Uppaal runs out of memory or continues without completing verification.

```
A[] R3_left.valid && R3_right.valid
```

Figure 61 – Pilot Flying Requirement R3 in Real-time Uppaal

Property R4 stating that the system starts with the primary (left) side as the pilot flying side can be stated in Uppaal as shown in Figure 62. This property states that if the global time *t* is one or less, the output *LPFS* is true. Since the initial value of the port *LPFS* is set to true, this is the expected behavior. This proof completes in 171 seconds.

```
A[] t <=1 imply LPFS == true
```

Figure 62 – Pilot Flying Requirement R4 in Real-time Uppaal

Finally, property R5 stating that the pilot flying side does not spontaneously change must be stated using the synchronous observer shown in Figure 63.

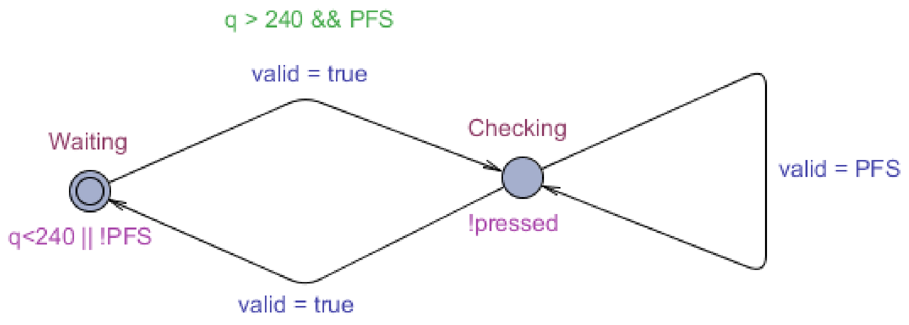


Figure 63 – Observer for Pilot Flying Requirement R5 in Real-time Uppaal

As with property R3, the observer starts in the location *Waiting* and transitions to location *Checking* when the Transfer Switch has not been pressed for more than 240 time units and this side is the pilot flying side, setting valid to true since the property has not yet been falsified. If the Transfer Switch is pressed while in location *Checking*, the observer transitions to the location *Waiting* since property R5 is only concerned with the system behavior when the Transfer Switch is not pressed. If the Transfer Switch is not pressed while in location *Checking*, valid is set to true if the side has remained the pilot flying side (*PFS* is true) and to false if the pilot flying side has spontaneously changed (*PFS* is false). The template for Figure 63 is named R5 and takes the single parameter *&PFS*. Process instances for the left and right sides are instantiated as *R5_left = R5(LPFS) and R5_right = R5(RPFS).* The query presented to Uppaal is shown in Figure 64. Unfortunately, the proof of property R5 does not complete, with Uppaal running out of memory or continues without completing verification.

.

```
A[] R5_left.valid && R5_right.valid
```

Figure 64 – Pilot Flying Requirement R5 in Real-time Uppaal

Proof times for requirements R1 through R5 for the Pilot Flying example with Uppaal running on an Intel® Core™ i73720QM processor running at 2.6 GHz with 4 GB of RAM are summarized in Table 8.

Table 8 – Pilot Flying Proof Times with Real-time Clocks in Uppaal

R1 (sec)	R2 (sec)	R3 (sec)	R4 (sec)	R5 (sec)
142	191	-	171	-

4.2.1.2 Pilot Flying Verification in Uppaal with Quasi-synchronous Clocks

Verification of the Pilot Flying example completed successfully for all the requirements using the general quasi-synchronous clock approach described in Section 3.2.2.2. As shown in Figure 65, when one of the clocks tick a variable *q* is incremented by one to record a step of execution. This allows measurement of the number of steps it takes for a property to be satisfied. The *Age* matrix described in Section 3.2.2.2 is declared to be an Uppaal meta variable, greatly reducing the time needed for verification. Without this change the verification runs out of memory. Incrementing the value *q* on each tick ensures that all possible transitions are considered even with the *Age* matrix declared as a meta variable.

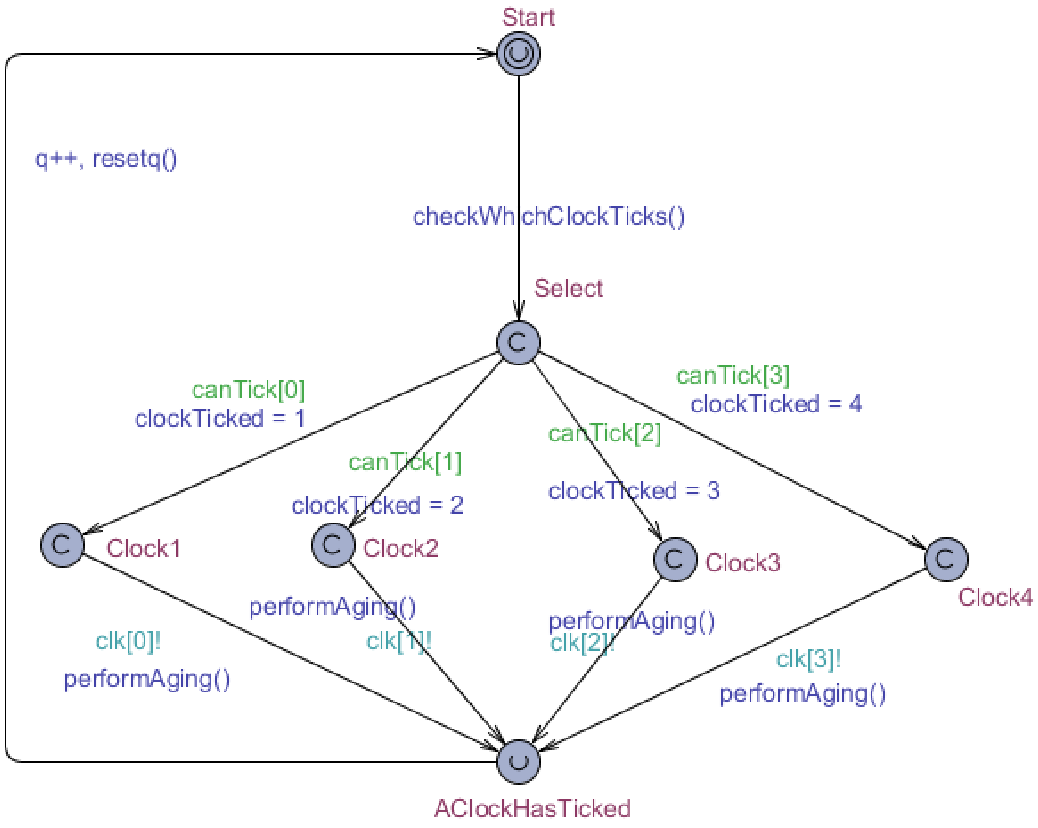


Figure 65 – Pilot Flying Quasi-synchronous Clock Constraints in Uppaal

In Uppaal the Transfer Switch is modeled as a global variable *TS* that is passed to both sides. To emulate this as a sampled input, we add the process shown in Figure 66. This process has three locations *Start*, *CheckTS*, and *TSPressed*. The transition from *Start* to *CheckTS*, randomly sets the value of its local Boolean variable *next_TS*, calls the local function *IsPressed* with argument *next_TS*, then sets the new value of *TS* to *next_TS*. The function *IsPressed* has two side-effects. It sets the local variable *pressed* to *(not TS and next_TS)* and, if *pressed* is true, it resets the global variable *q*. In this way, Uppaal treats *TS* as sampled input that changes randomly. The local variable *pressed* detects a rising edge of *TS* indicating whether the Transfer Switch was pressed. The variable *q* tracks the number of steps since *TS* was last pressed. *Pressed* and *q* are used only in stating properties to be checked, i.e., they are not used in the model itself. If *TS* is pressed, the process transitions to the state *TSPressed* where it waits for seven steps before transitioning to the *Start* state. This ensures that the value pressed must be true for at least seven steps, ensuring the non-Pilot Flying side sees the button press.[25] If *TS* is not pressed the process transitions to the *Start* state directly from the *CheckTS* state.

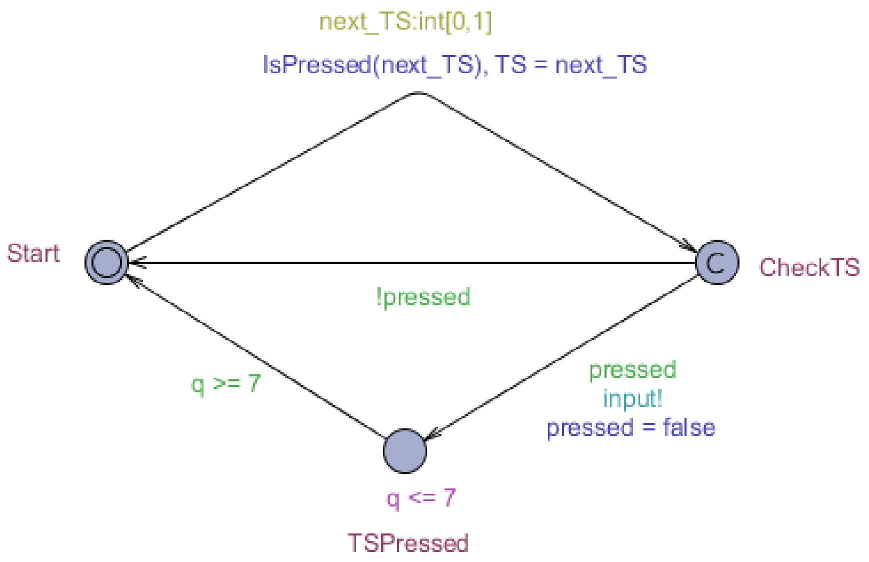


Figure 66 – Transfer Switch Input Process in Quasi-synchronous Uppaal

With a process defined to generate the Pilot Flying inputs, the next step is to formally state the Pilot Flying requirements. The first Pilot Flying requirement R1 from Section 3.3 states that at least one side is always the pilot flying side. The Uppaal query for this is shown in Figure 67. This property states that for all states and all paths (i.e., always henceforth) either the output *LPFS* is true or the output *RPFS* is true.

```
A[] LPFS || RPFS
```

Figure 67 – Pilot Flying Requirement R1 in Quasi-synchronous Uppaal

For this property to prove in Uppaal, the value of *q* must be periodically reset to zero, otherwise the state space grows without bound and Uppaal runs out of memory. The sooner this reset occurs, the faster the proofs complete, but the larger *q* is allowed to grow, the more confidence one can have that all important behaviors have been checked. For all the properties verified, the value of q was reset after 50, 500 and 1000 steps. As shown in Table 9, property R1 proves in seven to 26 seconds depending on when q is reset.

The formal statement of requirement R2 that both sides shall agree on the pilot flying side is shown in Figure 68. This states that for all states and paths, if the Transfer Switch has not

[25] This serves the same purpose as constraining Transfer Switch to remain true for at least seven steps in AGREE.

been pressed for 25 steps, the output *LPFS* will not equal output *RPFS*, i.e. the two sides will agree on who is the pilot flying side. As shown in Table 9, this property can be proven with Uppaal in seven to 26 seconds depending on when q is reset.

```
A[] q >= 25  imply LPFS != RPFS
```

Figure 68 – Pilot Flying Requirement R2 in Quasi-synchronous Uppaal

Stating Property R3 that pressing the Transfer Switch will change the pilot flying side requires the use of the synchronous observer processes shown in Figure 69.

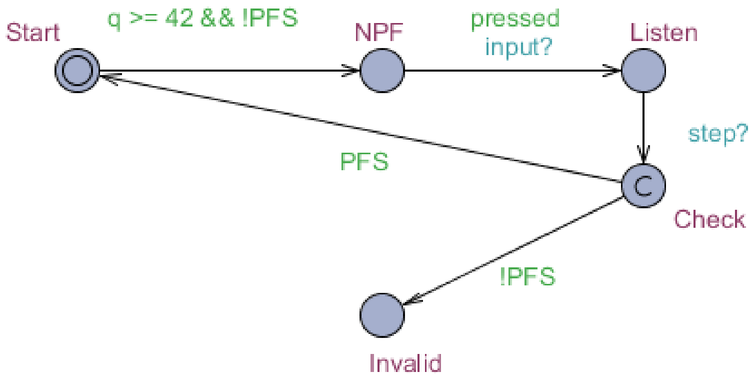


Figure 69 – Observer for Pilot Flying Requirement R3 in Quasi-synchronous Uppaal

The observer of Figure 69 transitions from location *Start* to location *NPF* whenever the Transfer Switch has not been pressed for more than 42 steps and the side is not the pilot flying side. If the Transfer Switch is not pressed while in location *NPF*, no change occurs since property R3 is only concerned with what happens when the Transfer Switch is pressed. If the Transfer Switch is pressed while in location *NPF*, the observer transitions to location *Listen*. When a *step?* event arrives indicating that the side has taken a step, the observer transitions to the location *Check*. If the side is the pilot flying side, the observer then returns to the *Start* location. Otherwise, it transitions to the *Invalid* location indicating that the property has been violated. The template for the process of Figure 69 is named R3. The actual query verified by Uppaal (shown in Figure 70) checks that the *Invalid* location is never reached.

```
A[] !R3L.Invalid
A[] !R3R.Invalid
```

Figure 70 – Pilot Flying Requirement R3 in Quasi-synchronous Uppaal

Property R4 stating that the system starts with the primary (left) side as the pilot flying side can be stated in Uppaal as shown in Figure 71.This property states that at initialization before the first clock tick, the output *LPFS* is true. This proof completes in seven to 24 seconds.

```
A[] age[0][0]==1 && age[1][0]==1 && age[2][0]==1 && age[3][0]==1
                         imply LPFS == true
```

Figure 71 – Pilot Flying Requirement R4 in Quasi-synchronous Uppaal

Property R5 stating that the pilot flying side does not spontaneously change can be stated using the synchronous observer shown in Figure 72.

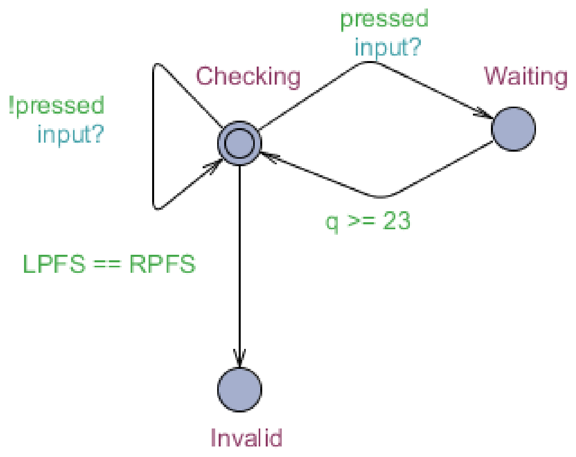


Figure 72 – Observer for Pilot Flying Requirement R5 in Quasi-synchronous Uppaal

The observer starts in the location *Checking* and remains at that location until the Transfer Switch is pressed. If both sides become the pilot flying side while in this location, the observer transitions to the *Invalid* location, indicating that a side has spontaneously changed its status. When the Transfer Switch is pressed, the observer transitions to the *Waiting* location, indicating that the system is in a transient state during which a side may spontaneously change. The observer remains in the *Waiting* location for 23 steps before returning to the *Checking* location. The template for Figure 72 is named R5 and the query presented to Uppaal is shown in Figure 73. The proof of property R5 completes in nine to 49 seconds.

```
A[] !R5.Invalid
```

Figure 73 – Pilot Flying Requirement R4 in Quasi-synchronous Uppaal

Table 9 below shows the time required to prove the Pilot Flying properties in Uppaal using quasi-synchronous clocks. The proofs were run on an Intel® Core™ i5-3320M processor running at 2.6 GHz with 4 GB of RAM.

Table 9 – Pilot Flying Proof Times with Quasi-synchronous Clocks in Uppaal

Property	q Reset at 50 Steps	q Reset at 500 Steps	q Reset at 1000 Steps
R1	7 seconds	15 seconds	26 seconds
R2	7 seconds	15 seconds	26 seconds
R3	5 seconds	9 seconds	15 seconds
R4	7 seconds	15 seconds	24 seconds
R5	9 seconds	29 seconds	49 seconds

4.2.2 Leader Selection Verification with Uppaal.

This section discusses the formal verification of the Leader Selection example with Uppaal. Section 4.2.2.1 discusses verification using real-time clocks and Section 4.2.2.2 discusses verification using quasi-synchronous clocks.

4.2.2.1 Leader Selection Verification in Uppaal with Real-time Clocks

The values chosen for the period and jitter of each clock in the Leader Selection example are shown in Table 10. These are easily seen to satisfy the quasi-synchronous constraints described in Section 3.2.1.

Table 10 – Leader Selection Clock Periods and Jitter for Uppaal Verification

Clock	Subsystem	Period	Jitter
CLK1	Node 1	46	4
CLK2	Node 2	50	0
CLK3	Node 3	54	2
CLK_CNB	Cross Node Bus	78	5

In the Leader Selection example, there are three inputs providing the measured health of each node as shown in Figure 18. These inputs are represented in Uppaal as three global variables, *Health1*, *Health2* and *Health3*. To emulate the system inputs, we instantiate three processes, passing each process one of the global health variables as the parameter *&Health*. The template for these processes is shown in Figure 74.

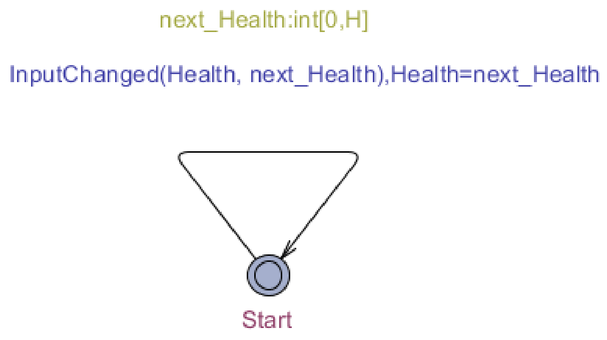


Figure 74 – Leader Selection Input Process for Real-time Uppaal

Each process has a single location *Start* with a single transition. In taking this transition, it randomly sets the value of its local integer variable *next_Health* to a value between 0 and *H*, calls the local function *InputChanged* with arguments *Health* and *next_Health*, then sets the new value of *Health* to *next_Health*. The function *InputChanged* sets a global clock q to 0 if the health changed on this step. In this way, Uppaal treats *Health1, Health2, and Health3* as sampled inputs that change randomly. The local variable *pressed* detects a change in any of the three inputs. The local clock *q* tracks the time since any of the three inputs was changed. It is used only in stating properties to be checked, i.e., it is not used in the model itself.

The first Leader Selection requirement R1 from Section 3.3.2 states that all nodes will agree on the leader. However, this property only holds when the system is in a quiescent state, The Uppaal query for this is shown in Figure 75. It states that for all states and all paths that if none of the inputs have changed for more than 100 time units, all nodes will agree on which node is the leader. This property can be proven with Uppaal in approximately 1 second with *Health* assigned a value ranging from [0-1], 14 seconds with *Health* assigned a value ranging from [0-2] and 152 seconds when the range for *Health* is [0-3].

```
A[] q > 100 imply (Leader1 = Leader2 && Leader2 = Leader3)
```

Figure 75 – Leader Selection Requirement R1 in Real-time Uppaal

The second Leader Selection requirement R2 states that the leader will always be the healthiest node. Again, this property only holds when the system is in a quiescent state. This is formulated as the three Uppaal queries shown in Figure 76. Since Requirement R1 should hold for q > 100, it does not matter which node's leader is used in the antecedent. Also note that ties are broken in favor of the node with the lower index. This property can be proven with Uppaal in approximately 1 second with *Health* assigned a value ranging from [0-1], 14 seconds with *Health* assigned to a value ranging from [0-2], 151 seconds when the range for *Health* is [0-3].

```
A[] q > 100 imply
     (Leader1 = 1 imply (Health1 >= Health2 && Health1 >= Health3))

A[] q > 100 imply
     (Leader2 = 2 imply (Health2 >  Health1 && Health2 >= Health3))

A[] q > 100 imply
     (Leader3 = 3 imply (Health3 >  Health1 && Health3 >  Health2)
```

Figure 76 – Leader Selection Requirement R2 in Real-time Uppaal

The third Leader Selection requirement R3 states that the leader will not change if the health of a node does not change. Specifying this property requires a way to identify when a leader has changed. Since Uppaal does not have a *pre* or *next* operator, this must be done by introducing three synchronous observers such as the one shown in Figure 77.

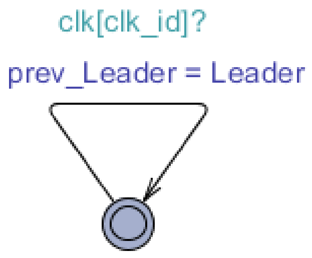


Figure 77 – Uppaal Synchronous Observer for Detecting Leader Change

The template of Figure 77 is passed the current *Leader* output of a node and the *clk_id* of that node's clock. Each time the node's clock ticks, it stores away the current value of the node's leader in its local variable *prev_Leader*. Three instances of this process are created, one for each node, named *prevLeader1*, *prevLeader2* and *prevLeader3* such that they execute prior to the process for their node. Requirement R3 can then be stated as shown in Figure 78. This property can be proven with Uppaal in approximately 1 second with *Health* assigned a value ranging from [0-1], 15 seconds with *Health* assigned to a value ranging from [0-2] and 160 seconds when the range for *Health* is [0-3].

```
A[] q >= 100 imply
  !prevLeader1.change && !prevLeader2.change && !prevLeader3.change
```

Figure 78 – Leader Selection Requirement R3 in Uppaal

Proof times for requirements R1 through R3 for the Leader Selection example with Health ranging from [0-3] with Uppaal running on an Intel® Core™ i73720QM processor running at 2.6 GHz with 4 GB of RAM are summarized in Table 11.

Table 11 – Leader Selection Proof Times with Real-time Clocks in Uppaal

R1 (sec)	R2 (sec)	R3 (sec)
152	151	160

4.2.2.2 Leader Selection Verification in Uppaal with Quasi-synchronous Clocks

The quasi-synchronous clocks for the Leader Selection example are implemented just as for the Pilot Flying example (Sections 3.2.2.2 and 4.2.1.2). The first Leader Selection requirement R1 from Section 3.3.2 states that all nodes will agree on the leader when the system is in a quiescent state. The Uppaal query for this is shown in Figure 79. It states that for all states and all paths that if none of the inputs have changed for more than 26 steps, all nodes will agree on which node is the leader. This property can be proven with Uppaal in approximately 140 to 200 seconds with *Health* ranging from [0-1], but for values of *Health* ranging from [0-2] the Uppaal model checker runs out of memory in about 3 seconds.

```
A[] q >= 27 imply
   (Node1.Leader == Node2.Leader && Node1.Leader == Node3.Leader)
```

Figure 79 – Leader Selection Requirement R1 in Quasi-synchronous Uppaal

The second Leader Selection requirement R2 states that the leader will always be the healthiest node. Again, this only holds when the system is in a quiescent state. This is formulated as three Uppaal queries as shown in Figure 80. Since Requirement R2 should hold for q > 26, it does not matter which node's leader is used in the antecedent. This property can be proven with Uppaal in approximately 167 to 177 seconds with *Health* ranging from [0-1] but for values of *Health* ranging from [0-2] the Uppaal model checker runs out of memory in about 3 seconds.

```
A[] q >= 27 imply (Node1.Leader == 1 imply
(Node1.Health1 >= Node2.Health2 && Node1.Health1 >= Node3.Health3))

A[] q >= 27 imply (  Node2.Leader == 2 imply
(Node2.Health2 > Node1.Health1 && Node2.Health2 >= Node3.Health3))

A[] q >= 27 imply (  Node3.Leader == 3 imply
(Node3.Health3 > Node1.Health1 && Node3.Health3 > Node2.Health2))
```

Figure 80 – Leader Selection Requirement R2 in Quasi-synchronous Uppaal

The third Leader Selection requirement R3 states that the leader will not change if the health of a node does not change. Specifying this property requires a way to identify when a leader has changed. This is done by creating three instances of the process of Figure 77 just as was done for the real-time verification of requirement R3. Requirement R3 can then be stated as shown in Figure 81. This property can be proven with Uppaal in approximately 164 to 180 seconds with *Health* ranging from [0-1] but for values of *Health* ranging from [0-2] the Uppaal model checker runs out of memory in about 3 seconds.

```
A[] q >= 27 imply
  !prevLeader1.change && !prevLeader2.change && !prevLeader3.change
```

Figure 81 – Leader Selection Requirement R3 in Quasi-synchronous Uppaal

Proof times for requirements R1 through R3 for the Leader Selection example with *Health* ranging from [0-1] with Uppaal running on Intel® Core™ i5-3320M processor running at 2.6 GHz with 4 GB of RAM are summarized in Table 12.

Table 12 – Leader Selection Proof Times with Quasi-synchronous Clocks in Uppaal

Property	q Reset at 50 Steps	q Reset at 500 Steps	q Reset at 1000 Steps
R1	140 seconds	171 seconds	200 seconds
R2	167 seconds	178 seconds	177 seconds
R3	164 seconds	172 seconds	180 seconds

5.0 CONCLUSIONS

The main goals of this project were to provide system designers with an intuitive modeling environment that 1) allows systems engineers to easily specify the high-level architecture and synchronization logic of quasi-synchronous systems using widely available system engineering notations and tools, and 2) integrates and enhances innovative formal verification tools to provide system engineers with immediate feedback on the correctness of their designs.

To provide a realistic path for technology transfer, system developers can create high-level models of the system architecture and synchronization logic using the Enterprise Architect SysML [7] modeling environment enhanced with a SysML profile for quasi-synchronous systems. Translators import these models into the OSATE development environment for AADL. Component behaviors specified as SysML state machines are translated into AGREE and Behavior Annex specifications within AADL.

Verification of the AADL model supplemented with the AGREE annexes can be performed directly using the AGREE tool [10], [11], where AGREE can be configured to invoke either the Kind [12] or jKind SMT-based model checker. Verification of the AADL model supplemented with the Behavior Annexes can be performed using the Uppaal model checker [13] for timed automata by first invoking a translator that converts the AADL and Behavior Annex specifications into an Uppaal model. This model can then be verified using the graphical user interface provided with the Uppaal tool.

Four examples of quasi-synchronous systems were created and verified: the Pilot Flying example, the Leader Selection example, the Active Standby example, and the WBS example. All of these are based on actual examples seen in industry. The WBS example is derived from an accident report in which a commercial air transport aircraft lost all braking capability on landing.

Another significant accomplishment was formalizing the notion of quasi-synchrony and relating it back to real-time parameters such as the period, jitter and offset of each clock. Logic was developed in AGREE and in Uppaal to constrain clocks to be synchronous, quasi-synchronous, or asynchronous, allowing a user to easily verify their design assuming any of these models of clock synchronization. In addition the notion of quasi-synchrony described in [4] was generalized and stated formally so that arbitrary relations between the clocks can be specified.

We developed two approaches for the verification of quasi-synchronous systems in Uppaal. The first approach takes advantage of the real-time capabilities of Uppaal and models clocks with actual periods and jitter that met the quasi-synchronous constraints. Though this is not as general as verifying all possible quasi-synchronous assignments of period and jitter as was done with Kind, it has the advantage of producing actual values for how long it takes after the system is stimulated before a property holds. It is also likely that system developers are more likely to accept an analysis based on the actual parameters of their system than one based on the abstraction of quasi-synchrony.

The second approach constrains the clocks so that they meet the quasi-synchronous constraint. On each step, one clock is selected to tick, but no clock is allowed to tick in violation of the quasi-synchronous constraint. This was implemented in a general way that allows the user to specify any version of n/m quasi-synchrony, where $0 < m < n$. Since only one clock can tick on each step, this is actually a subset of true quasi-synchrony. In fact, it is the maximally asynchronous subset of quasi-synchrony. However, this is sufficient to compare the effectiveness of the two approaches.

Verification of the four examples was conducted using both AGREE/Kind and Uppaal. The properties of all four models were ultimately verified using AGREE/Kind. While the Pilot Flying and Leader Selection examples were verified using Uppaal, Uppaal was unable to complete the verification of Active Standby and WBS examples. As discussed later, we attribute this to a mismatch between the strengths of Uppaal and the specific challenges posed by quasi-synchronous systems. Constraining the order in which the clocks of the distributed nodes can tick using the quasi-synchronous abstraction eliminates the need for real-time analysis. In other domains where there is a need for real-time analysis, Uppaal would have clear advantages over the use of AGREE/Kind.

Currently, the greatest shortcoming of the tools is that the formal verification can take several minutes or even hours. Systems with more clocks take longer to verify and the time appears to grow exponentially with the number of clocks. We were not able to develop strategies to reduce the verification time solely by exploiting the properties of the quasi-synchronous constraints. A direction for future research is to investigate alternate strategies for model checking, such as the use of Property Directed Reachability (PDR) [40], [41] or partial-order reduction [42].

5.1 Observations

An underlying hypothesis of this project was that the development of distributed agreement protocols could be simplified by exploiting the quasi-synchronous clocks found in most actual systems. This was motivated by the observation that many engineers achieve distributed agreement by inserting wait states rather than by implementing the hand-shaking protocols necessary for fully asynchronous systems.

This hypothesis does hold – it is possible to insert wait states that exploit the timing constraints of quasi-synchronous clocks to develop distributed agreement protocols that can be proven correct. These protocols are simpler than those required for fully asynchronous systems in that they do not have to have to exchange acknowledgements as required by hand-shaking protocols.

At the same time, the development of distributed agreement protocols, even under an assumption of quasi-synchrony, remains a difficult task. The interleaving of clocks allowed by the quasi-synchronous constraints still results in a significant increase in the number of reachable states over that seen for a synchronous system. While formal verification can determine whether a wait state is too short or has not been inserted in the correct place, it does not automatically identify where wait states need to be inserted or how long they need to be. Debugging a counter-example may provide valuable insight into where wait states are needed and how long they need to be, but designing distributed agreement protocols still requires considerable human insight.

On the other hand, providing developers with immediate feedback on the correctness of their designs is an enormous improvement on the current state of affairs. Formal verification identified numerous errors in early versions of the protocols and made explicit assumptions, such as the duration of button presses, that the developers are depending on. The WBS example was particularly illustrative. In that example, the failure of the Braking System on an air transport class aircraft was traced to an error in sampling button presses by the MON and CON units that executed at the same period but that were not synchronized. The cause of this error was easily found using formal verification of quasi-synchronous systems, but had been missed in the rigorous review and testing required of civil avionics systems. This analysis also produced a different counter-example whose cause was not described in the accident report.

Without formal verification, developers must rely on reviews and testing, both of which are inadequate for the verification of distributed systems. This almost certainly results in the deployment of systems that appear to work but can still fail due to latent design errors. Quasi-synchrony also appears to be the simplest approximation of realistic systems short of implementing a synchronous system. In previous work with the University of Illinois we developed an approach called Physically Asynchronous/Logically Synchronous (PALS) that implements logical synchrony over a physically asynchronous platform [34]. Later work showed that even for simple distributed agreement protocols PALS reduced the reachable state space almost three orders of magnitude over that for a fully asynchronous system [35]. Experiments conducted on the example models verified here confirm that formal verification for a synchronous system is considerably faster and easier than for a quasi-synchronous or asynchronous systems. This confirms that approaches to achieve logical synchrony such as PALS or Loosely Time-Triggered Architectures (LTTA) [39] have merit. Unfortunately, it is difficult to convince developers of this when existing protocols verified through testing appear to work most of the time.

The other alternative is to develop distributed agreement protocols for fully asynchronous systems, which will work for either synchronous or asynchronous systems. Many such protocols are described in the literature [1], [2], they do not seem to be widely known by system developers.

At this time, the Kind model checker and the Lustre language appear to be a better fit for the verification of quasi-synchronous systems than the Uppaal model checker and language. It is easier to introduce bounded asynchrony at the higher levels of a model in a synchronous language such as Lustre than it is to introduce synchrony at the lower levels of a model in an asynchronous language such as Uppaal. This makes Lustre a more natural target language for the translation of the globally asynchronous, locally synchronous AADL models of interest here. It is also easier to specify the properties of interest in the Lustre language than in the temporal logic of Uppaal. The availability of the *pre* operator in Lustre to refer to the previous state of a variable is more appropriate for stating the bounded safety properties than the *eventually* liveness operator of Uppaal. A *next-state* operator in Uppaal would address this need, but does not appear to be offered at this time. As a result, several properties could be stated in Uppaal only through the use of synchronous observers. Finally, constraining the order in which the clocks of the distributed nodes can tick using the quasi-synchronous abstraction eliminated the need for real-time analysis. However, Uppaal would have had a clear advantage analyzing systems with a large number of concurrent, asynchronous components, particularly if real-time properties were of interest.

5.2 Future Directions

There remain several possible directions for further research. The general version of quasi-synchrony is being implemented in the AGREE tool. It would be interesting to investigate other forms of quasi-synchrony than 2/1 quasi-synchrony. For example, do higher values of n/m reduce or increase the time needed for formal verification? Are other forms of quasi-synchrony, e.g., 7/4 quasi-synchrony, of practical use?

If some of the clocked nodes in a system are independent of each other (i.e., do not exchange messages or share memory), it would seem that the order in which their clocks execute are irrelevant. Can this be exploited to reduce the time needed for verification of such sparsely connected quasi-synchronous systems?

Finally, it may be possible that different types of model checkers may be more appropriate for the verification of quasi-synchronous systems. Since many quasi-synchronous systems have a relatively small number of system states, Binary Decision Diagram (BDD) based model checkers may actually be more appropriate than SMT-based model checkers for some systems. Still other directions to explore include the use of Property Directed Reachability [40], [41] or partial-order reduction [42].

REFERENCES

[1] Nancy Lynch, Distributed Algorithms, Morgan Kaufmann Publishers, Inc: San Francisco, CA, 1996.

[2] Gerard Tel, Introduction to Distributed Algorithms, Second Edition, Cambridge University Press: Cambridge, UK, 2000.

[3] Paul Caspi, Christine Mazuet and Natacha Reynaud Paligot, About the Design of Distributed Control Systems: The Quasi-Synchronous Approach, in Proceedings 20th International Conference on Computer Safety, Reliability, and Security (SAFECOMP 2001), pg. 215-226, Springer-Verlag: London, UK, 2001.

[4] Nicolas Halbwachs and Louis Mandel, Simulation and Verification of Asynchronous Systems by means of a Synchronous Model, in Proceedings of 6th International Conference on Application of Concurrency to System Design (ACSD'06), Turku, Finland June 28-30, 2006.

[5] The Mathworks, Simulink Product Description, http://www.mathworks.com.

[6] Esterel Technologies, SCADE Suite Product Description, http://www.estereltechnolgies.com.

[7] Sanford Friedenthal, Alan Moore, and Rick Steiner. A Practical Guide to SysML, Elsevier: New York, 2012.

[8] Sparx Systems Enterprise Architect homepage, http://www.sparxsystems.com.au/.

[9] Peter H. Feiler and David P. Gluch, Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language, The SEI Series in Software Engineering, Addison-Wesley, 2012.

[10] Darren Cofer, Andrew Gacek, Steven P. Miller, Michael Whalen, Brian LaValley, and Lui Sha, Compositional Verification of Architectural Models, in Proceedings 4th NASA Formal Methods Symposium, Norfolk VA, April 2012.

[11] Michael Whalen, Andrew Gacek, Darren Cofer, Anitha Murugesan, Mats P.E. Heimdahl, Sanjay Rayadurgam, Your "What" is my "How": Iteration and Hierarchy in System Design, IEEE Software, Vol. 30, Issue 2, November, 2012.

[12] Kind homepage, http://clc.cs.uiowa.edu/Kind/.

[13] Uppaal homepage, http://www.uppaal.org/.

[14] SAE Standard AS5506, Architecture Analysis and Design Language (AADL), 2006.

[15] SAE Standard AS5506/2, Architectural Analysis and Design Language (AADL) Annex Volume 2, Annex D: Behavior Modeling Annex, 2011.

[16] AADL-BA-FrontEnd, http://wiki.sei.cmu.edu/aadl/index.php/AADL-BA-FrontEnd.

[17] AADL OSATE homepage, http://www.aadl.info/aadl/currentsite/tool/osate.html.

[18] Steven P. Miller, Michael Whalen, and Darren Cofer, Software Model Checking Takes Off, Communications of the ACM, Vol. 33, ISS 2, February, 2010.

[19] George Hagen and Cesare Tinelli, Scaling up the Formal Verification of Lustre Programs with SMT-based techniques, in Proceedings of the 8th International Conference on Formal Methods in Computer-Aided Design (FMCAD'08), Portland, Oregon. 2008.

[20] George Hagen, Verifying safety properties of Lustre programs: an SMT-based approach, PhD dissertation, The University of Iowa, 2008.

[21] Temesghen Kahsai and Cesare Tinelli, PKIND: A Parallel k-induction Based Model Checker, in Proceedings of the 10th International Workshop on Parallel and Distributed Methods in verification (PDMC'11), EPTCS, 2011.

[22] P. Caspi, D. Pialiud, N. Halbwachs and J. Plaice, LUSTRE: a Declarative Language for Programming Synchronous Systems, in 14th Symposium on Principles of Programming Languages, pages 178–188, 1987.
[23] Robert Nieuwenhuis, Albert Oliveras and Cesare Tinelli, Solving SAT and SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T), The Journal of the ACM, 53(6), pg. 937–977, November 2006.
[24] Clark Barett, R. Sebastiani, S. Seshia and Cesare Tinelli, Chapter on Satisfiability Modulo Theories, in A. Biere, H. van Maaren and T. Walsh editors, Handbook on Satisfiability. IOS Press, February 2009.
[25] Leonardo de Moura and Nikolaj Bjorner, Satisfiability Modulo Theories: Introduction and Applications, Communications of the ACM, September, 2011.
[26] Clark Barett, Aaron Stump, and Cesare Tinelli, The SMT-LIB Standard Version2.0, September 9, 2012, available at http://www.smtlib.org.
[27] Clark Barett, and Cesare Tinelli, CVC3, in Proceedings of the 19th International Conference on Computer Aided Verification (CAV'070), Springer, 2007.
[28] Clark Barett, Christopher Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli, CVC4, in Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11), Springer, 2011.
[29] Temesghen Kahsai, Yeting Ge and Cesare Tinelli, Instantiation-Based Invariant Discovery, in Proceedings of the 3rd NASA Formal Methods Symposium (NFM'11). Springer, 2011.
[30] Rajeev Alur and David L. Dill. The Theory of Timed Automata, TCS, 126(2), 1994.
[31] Xiaowan Huang, Anu Singh, Scott A. Smolka, Using Integer Clocks to Verify Clock-Synchronization Protocols, Innovations in Systems and Software Engineering, Volume 7 Issue 2, pg. 119-130, June 2011.
[32] R. Grosu, X. Huang, S. A. Smolka, W. Tan, and S. Tripakis, Deep Random Search for Efficient Model Checking of Timed Automata, in Proceedings of the 13th Monterey Conference on Composition of Embedded Systems, pages 111–124, Berlin, Heidelberg, 2007. Springer-Verlag.
[33] Steven P. Miller, Mike Whalen, Dan O'Brien, Mats P.E. Heimdahl, and Anjali Joshi, A Methodology for the Design and Verification of Globally Asynchronous/Locally Synchronous Architectures, NASA Contractor Report NASA/CR-2005-213912, Sept. 2005. Available at http://shemesh.larc.nasa.gov/fm/papers/Miller-CR-2005-213912-Methodology-GALS.pdf.
[34] Steven P. Miller, Darren Cofer, Lui Sha, Jose Meseguer, Abdullah Al-Nayeem, Implementing Logical Synchrony in Integrated Modular Avionics, in Proceedings of the 28th Digital Avionics Systems Conference (DASC 2009), Orlando, Florida, October 27-29, 2009.
[35] Abdullah Al-Nayeem, Lui Sha, Darren Cofer, Steven P. Miller, Pattern-based Composition and Analysis of Virtually Synchronized Real-Time Distributed Systems, in Proceedings of the Third International Conference on Cyber-Physical Systems (ICCPS), Beijing, China, April 17-19, 2012.
[36] Cindy Eisner and Dana Fisman, A Practical Introduction to PSL, Springer Science + Business Media, 2006.
[37] John Rushby, A Comparison of Bus Architectures for Safety-Critical Embedded Systems, NASA Contractor Report CR-2003-212161, September 2001.

[38] Civil Aviation Accident and Incident Investigation Commission (CIAIAC), Spain, Technical Report: Accident occurred on 21May 1998 to Aircraft Airbus A-320-21 Registration G-UKLL At Ibiza Airport, Balearic Islands, http://www.fss.aero/accident-reports/dvdfiles/ES/1998-05-21-ES.pdf.

[39] S. Tripakis, C. Pinello, A. Benveniste, A. Sangiovanni-Vincent, P. Caspi, and M. Di Natale, Implementing Synchronous Models on Loosely Time Triggered Architectures, IEEE Transactions on Computers, vol. 57, no. 10, pg. 1300-1314, 2008.

[40] Aaron R. Bradley, SAT-Based Model Checking without Unrolling, in R. Jhala and D. Schmidt (Eds): VMCAI 2011, LNCS 6538, pg. 70-87, 2011.

[41] Niklas Een, Alan Mischenko and Robert Brayton, Efficient Implementation of Property Directed Reachability, in Proceedings of Formal Methods in Computer-Aided Design (FMCAD), Oct. 30 – Nov. 2, 2011.

[42] Gerard J. Holzmann, The Spin Model Checker, Addison-Wesley, 2004.

LIST OF ACRONYMS

AADL	Architecture Analysis and Design Language
AFDX	Avionics Full-Duplex Switched Ethernet
AGREE	Assume Guarantee Reasoning Environment
BA	Behavior Annex
BDD	Binary Decision Diagram
BMC	Bounded Model Checking
BSCU	Brakes & Steering Control Unit
DARPA	Defense Advanced Research Projects Agency
DFA	Deterministic Finite Acceptor
FCS	Flight Control System
FGS	Flight Guidance System
IDE	Integrated Development Environment
LTTA	Loosely Time-Triggered Architectures
NASA	National Aeronautics and Space Administration
OMG	Object Management Group
OSATE	Open Source AADL Tool Environment
PALS	Physically Asynchronous/Logically Synchronous
PDR	Property Directed Reachability
PFCS	Primary Flight Control System
PFS	Pilot Flying Side
SAE	Society of Automotive Engineers
SCADE	Safety Critical Application Development Environment
SMT	Satisfiability Modulo Theories
SysML	System Modeling Language
UML	Unified Modeling Language
TS	Transfer Switch
TTA	Time-Triggered Architecture
WBS	Wheel Braking System
XML	Extensible Markup Language

APPENDIX A: TRANSLATION FROM AADL TO KIND

A translator from AADL and Behavior Annex specifications to the Lustre [22] language was developed in Phase 1 of this project. The Kind mode checker will input the Lustre language and generate proof obligations in the standard SMT-LIB format [26], or in SMT solver-specific formats, and uses an SMT solver to discharge them. This translator was later superseded by the development of the AGREE translator, but the original AADL to Lustre translator has been included with the other tools. This appendix describes how the translator converts AADL and Behavior Annex specifications into Lustre.

We have implemented a translator from a declarative AADL model where component behavior is specified with the Behavior Annex of AADL to a program in the fragment of the synchronous data-flow language Lustre that is accepted by the Kind model checker. Following the translation of the components, a constraint is generated to make all clocks pairwise quasi-synchronous and invariants to be verified are inserted from a separate file together with auxiliary nodes used in the specification of the invariants. In order to increase performance of the verification, the translator recognizes frequently used components of certain packages by their name and inserts optimized Lustre nodes from a library instead of invoking the generic translation.

The translator takes a declarative AADL model as input and it is not necessary to instantiate the declarative model to an instance model as required by other tools. In an instance model all connections are flattened to physical connections between subcomponents. This allows reasoning on the physical level, but we are also interested in structural properties of the system. Further, in order to support modular reasoning, it is necessary to preserve the hierarchy in the declarative model and in particular the hierarchical structure of connections between components.

A.1 Translation of AADL Models

A declarative AADL model consists of *component types* and *implementations* of component types. A component type defines externally observable features, in our case just input and output data ports. An implementation of a component type C is a composition of *subcomponents* of types C_j, which are in turn either component types or implementations, and *connections* between the input and output ports of C_j and the input and output ports of C. Due to limitations of the Behavior Annex, some component implementations contain *data subcomponents* for which properties can be set, which is not possible for variables in a Behavior Annex. Such data subcomponents are not used for any other purpose and in particular are not accessible from other components.

There can be several implementations of the same component type, a component always refers to one particular implementation. Both component types and implementations can have a Behavior Annex that specifies the input-output relation of the component as a finite state machine. The Behavior Annex of an implementation overrides a Behavior Annex of its type.

The translation takes as input a component implementation C^I in a declarative model and creates a Lustre program that models the behavior of C^I its subcomponents and their subcomponents.

Each component type or implementation must have a clock associated with it through the property *QS_Properties::Clock_Name*. Clocks of the same name are identical; the property is inherited, so that components that do not explicitly set a clock, run on the clock of their containing component.

A.2 Component Types

The only supported sections of a component type are *features* and *annexes*, no distinction is made between different component categories such as systems or threads. A feature may only be a *data port*; event ports or event data ports are not supported. The annex must only be a *behavior_specification*.

We view a component type $C = (F_{in}, F_{out}, B)$ as consisting of

- a set of input data ports F_{in},
- a set of output data ports F_{out}, and
- an optional behavior annex B.

A port is in both F_{in} and F_{out} if it is a bidirectional port.

A component type C with n input and m output ports is translated to a Lustre node with the signature

```
node C (i₁ : tᵢ₁, …, iₙ : tᵢₙ) returns (o₁ : tₒ₁, …, oₘ : tₒₘ);
```

where for $1 \leq j \leq n$ and $1 \leq k \leq m$ each i_j and o_k is a stream variable corresponding to a port $i_j \in F_{in}$ and $o_k \in F_{out}$, respectively, andt_{ij} and t_{ok} are the Lustre types corresponding to the types of i_j and o_k, respectively.

If the component type is one of the defined component types, the node body is taken from there.

If the component type has a behavior annex B, the body of the node is the translation of B described below.

A component that is not a defined component and does not have a behavior annex must not occur as a subcomponent of a component implementation.

A.2.1 Component Implementations

Supported sections of a component implementation are *subcomponents*, *connections* and *annexes*, which must be a *behavior_specification*.

A component implementation $C^{I\,=}\,(S, L, B)$ of the component type C consists of

- a set of subcomponents S, which are either component implementations or component types,
- a set of connections L, and
- an optional behavior annex B.

Notably, a component implementation does not specify input or output ports, since those are already defined in its type that is shared by all implementations.

The starting point of a connection $p_{source} \rightarrow p_{target}$ in L can be an input port of the component type C, an output port in F_{out} of some component type $C_j \in S$ of a subcomponent or an output port in F_{out} of the component type C_j of some component implementation $C^I_j \in S$. In the analogous way, the endpoint of a connection p_{target} is either an output port in F_{out} of the component type C, an input port in F_{in} of some component type $C_j \in S$ or in F_{in} of the component type C_j of some component implementation $C^I_j \in S$.

The component implementation C^I must connect to each input port of the type C_j of each subcomponent $C_j \in S$ or $C^I_j \in S$ either one of the input ports of C or an output port of the type C_k of a subcomponent $C_k \in S$ or $C^I_k \in S$. If the component implementation C^I or its type C has a Behavior Annex, this restriction is relaxed so that instead of a connection to an input port of type C_j of a subcomponent the port may also be assigned to the action part of a transition.

There must be at most one connection to each input port of a subcomponent C_j and to the output port of the component implementation C^I; the number of connections from an output port of a subcomponent or from the input ports of the component implementation is not restricted. The data types of connected ports must match, however, this is not checked in the translation.

Let the component implementation C^I contain k subcomponents in S. If the j-th subcomponent in S is a component type, let C_j be this component type, otherwise if the j-th subcomponent of S is a component implementation C^I_j, let C_j be its component type. Let N_j be the node translated from the j-th subcomponent of C and let $(t^j_{om1}, \dots, t^j_{omj})$ be the type of its m_j-tuple output.

The component implementation C^I is translated to a node with a signature identical to the signature of its component type C as described above. In addition, the node contains m_j stream variables for each subcomponent C_j or C^I_j with m_j output ports.

```
node $C^I$ ($i_1$ : $t_{i1}$, …, $i_n$ : $t_{in}$) returns ($o_1$ : $t_{o1}$, …, $o_m$ : $t_{om}$);
var
$s^1_1$ : $t^1_1$; …; $s^1_{m1}$ : $t^1_{m1}$;
⁞
$s^k_1$ : $t^k_1$; …; $s^k_{mk}$ : $t^k_{mk}$;
let
($s^1_1$, …, $s^1_{m1}$) = $N^1$($p^1_1$, …, $p^1_{n1}$);
⁞
($s^k_1$, …, $s^k_{mk}$) = $N^k$($p^k_1$, … $p^k_{nk}$);
tel
```

Each p_i^j is identified with a stream variable in the following way. If there is a connection from the u-th output port of the v-th subcomponent, then $p_i^j = s^v_u$. Otherwise, if the connection source is the u-th input port of the component implementation, then $p_j^i = i_u$. If there is no connection to an input port of a subcomponent, the input port must be assigned by a transition of the Behavior Annex of that subcomponent and p_i^j is identified with the stream created there, see below.

For each connection from the u-th output port of the v-th subcomponent to the j-th output port of the component, an equation

```
$o_j = s^v_u$
```

is added to the node.

A.3 Properties

An AADL model may include user-defined property sets and annotate component types and implementations or specific instances in a model with properties defined in property sets. Properties are inherited by subcomponents if specified in the property set with the keyword *inherit* and can be overridden by subcomponents if not forbidden in the property set with the keyword *constant*.

We define a property set *QS_Properties* containing specific properties of quasi-synchronous models. The translator recognizes the not constant and inheriting *Clock_Name* property and requires it to be set for every component. The property set may be extended further to accommodate model-specific properties.

The translator also recognizes the property *Data_Model::Initial_Value* of data subcomponents. This property is defined in the Data Modeling Annex of AADL standard. It is used to initialize variables in state machines in the Behavior Annex as discussed below.

Each property that is read in the Behavior Annex of a component type or implementation or by one of its subcomponents is added as an input to the Lustre node. In particular, since every component must define the *QS_Properties::Clock_Name* property, every Lustre node *N* translated from a component type or implementation has its signature extended to

```
node N (…, QS_Properties_Clock_Name : bool) returns (…);
```

where the property, which is in fact of type string, is translated to a Boolean input parameter.

If a component implementation C^I contains a subcomponent *D*, a property that is an input parameter to *D* must be set when the node of C^I calls the node of *D*. If the property of *D* is set for the particular subcomponent, the specified value is passed to the node of *D* as its input parameter. If the property is inherited, the value of the property of the component implementation C^I is propagated. The value of the property of C^I is itself an input to the node of C^I and the node of *D* is called with this value. Otherwise the input parameter is set to a default value.

The *Clock_Name* property is inherited and used by every node, but never set to a value. A component implementation thus has one input for each clock it or one of its subcomponents uses, the topmost component implementation has all clocks of the model as inputs.

A.4 Behavior Annex

A Behavior Annex describes the behavior of a component or a component implementation as a finite state machine and consists of three sections, defining *variables*, *states* and guarded, prioritized *transitions* between states, respectively. If a component implementation contains a data subcomponent, it can be treated in the same way as a variable. Transition actions are assignments to variables, data subcomponents or output ports and are executed simultaneously. The translator only supports sets of assignment actions, sequences of assignment actions are treated as sets, and, in particular, loops and conditional blocks are not supported.

A BA occurs as part of a component type *C* or implementation C^I and a transition can read the input ports of *C* and assign to the output ports of *C*. In addition, a transition in the BA of a component implementation C^I can read the output ports and assign to the input ports of its subcomponents, as well as read and assign to its data subcomponents.

We view a BA as a tuple $B = (S, T, V, I)$, where

- $S = \{1, \dots, n\}$ is the set of states,
- *T* is the set of transitions,

- V is the set of typed state variables $v: R$, where R is a type, and
- $I \in S$ is the initial state.

We expect the state machines to be continuous: a final or complete state will never be reached.

A transition is a tuple $(s, s', p, F, A) \in T$, where

- $s \in S$ and $s' \in S$ are the source and the target state, respectively,
- $p \in \mathbb{N}$ is the priority of the transition,
- F is the guard of the transition, a Boolean expressions over the input ports of the component type C and the output ports of its subcomponents, and
- A is the set of assignments to state variables, data subcomponents or output variables, seen as a set of pairs (v, a), where v is a state variable $v: R \in V$, an output port in F_{out} of the component type C or, if the BA is of a component implementation C^I, an input port in F_{in} of some subcomponent of C^I. The assigned value a is of type R or the type of the port, respectively.

In the node of the component type C or implementation C^I that contains the Behavior Annex B we create a stream

```
state : subrange [1, n] of int
```

for the current state of the transition system, where n is the number of states in S. The syntax *subrange* is an extension to Lustre that results in the model checker assuming the variable to be in the given range and to treat the variable as a mode variable to which some heuristics are applied.

Further, we create a stream

```
t : bool
```

for each transition $t \in T$, a stream

```
v : R
```

for each variable $v: R \in V$, where R is the type of v, and a stream

$\mathtt{p^j_i : t^j_{oi}}$

if there is some assignment to the i-th input port of the j-th subcomponent of the component implementation C^I.

Together the streams model the state of the finite state machine as follows. The state machine is in state *state* and is executing the transition t whose corresponding stream is true. The guard of a transition reads the values of the input ports in the current state and the values of local variables in the previous state. Hence each occurrence of a local variable p_i^j is under a *pre* operator. The local variables and data subcomponents are set based on the assignment action in the currently executed transition, again from values of input ports in the current state, local variables and data subcomponents in the previous state.

Every node of a component type or component implementation has an input parameter that corresponds to its clock. In the following, let *clk* be this Boolean input parameter.

For each $s \in S$ let $T_s = t_{s1}, \ldots, t_{sk_s}$ be the sequence of the k_s transitions $(s, s'_{si}, p_{si}, F_s, A_{si}) \in T$ with the same source state s, ordered by their priorities such that t_{s1} is the transition out of s with the highest priority. For each $t_{si} \in T_s$ and each $s \in S$ we define

```
ts1 = clk and (1 -> pre(state) = s) and Fs1;
```

and

```
tsi = clk (1 -> pre(state) = s) and Fsi and not ts1 and … and not ts(i-1);
```

for $i > 1$. For each $s' \in S$ let $T'_{s'} = t'_{s'1}, \ldots, t'_{s'ms}$ be the sequence of the $m_{s'}$ transitions $(s_{s'i}, s', p_{s'i}, F_{s'i}, A_{s'i}) \in T$ with the same target state s', then we define

```
state = if t'11 or … or t'1m1 then 1
        else …
        else if t'n1 or … or t'nmn then n
        else (1 -> pre(state));
```

where I is the initial state.

For each variable $v: R \in V$ and each input port v in F_{in} of a subcomponent of the component implementation C^I that occurs in some assignment action, let $A_v = \{(t_1, a_{t1}), \ldots, (t_p, a_{tp})\}$ be the set of p pairs where t_i is a transition $(s_i, s'_i, p_i, F_i, A_i) \in T$ and $(v, a_{ti}) \in A_i$. We define for each $v: R \in V$

```
v = if t1 then at1
    else …
    else if tp then atp
    else (vi -> pre(v));
```

The value v_i is the initial value of the variable v. Since local variables in the BA cannot be annotated with the *Data_Model::Initial_Value* property, a local variable that is meant to have a specific initial value must be turned into a data subcomponent instead. In this way, if v is a data subcomponent, v_i is the value of the property if it is set, otherwise, if the property is not set or v is a local variable of the BA, v_i is 0 for integers and false for Booleans.

The same applies to a variable v that is read in assignment actions or guards. The state machine refers to the value of the variable in the previous step, hence the *pre* operator must be guarded with the -> operator. Every occurrence of a variable v is translated to the expression $(v_i \text{-> } pre(v))$, where v_i is as above.

A BA must be written such that at least one transition applies in any state. However, it may be the case that the guards of two transitions of equal priority out of the same state are satisfied, in which case one is chosen non-deterministically. The above translation assumes a fixed order of priorities and must be refined to account for the required non-determinism.

A.5 Types

The translation currently assumes all input and output data ports to be Boolean or integer. No structured types are supported at the moment.

A.6 Predefined Components

The following components have defined semantics and are translated to the respective Lustre nodes without the need to specify their behavior in the AADL model.

A.6.1 Signals::Rise: A rising edge detector

AADL

```
system Rise
features
   I: in data port Base_Types::Boolean;
   O: out data port Base_Types::Boolean;
end Rise;
```

Lustre

```
node Signals_Rise (I : bool; clk : bool)
returns (O : bool);
var pre_I: bool;
let
  pre_I = (clk => I) and (not clk => (false -> pre(pre_I)));
  O = false -> not pre(pre_I) and I;
tel;
```

A.7 Quasi-synchronous Constraint Generation

Based on the DFA of Figure 4 we can define the following Lustre node

```
node qs_dfa (p, q : bool) returns (ok : bool);
var r : int;
let
  ok = not (((0 -> pre r) = 2 and p) or ((0 -> pre r) = -2 and q));
  r = if p and q then 0
      else if p then (if (0 -> pre r) < 0 then 1 else ((0 -> pre r)) + 1)
      else if q then (if (0 -> pre r) > 0 then -1 else ((0 -> pre r)) - 1)
      else (0 -> pre r);
tel;
```

The inputs *p* and *q* are two clocks, the output *ok* is *true* if the clocks are running quasi-synchronous. The variable *r* can be understood as the relative advance of clock *p* over *q*, it ranges between -2 and 2. One can map the values of r to the states of the DFA by realizing that *r=0* corresponds to the state *0*, *r=1* to *1P*, *r=2* to *2P*, *r=-1* to *1Q* and *r=-2* to *2Q*. The transitions out of *2P* and *2Q* that are not accepted lead to *r=3* and *r=-3*, respectively and result in the output *ok* becoming *false*.

Since all clocks have to be pairwise quasi-synchronous, we create a node *calendar* with all clocks clk_1, …, clk_n of the model as input parameters and a single output parameter *ok* defined as the conjunction of calls to the *qs_dfa* node for all pairs of clocks (clk_i, clk_j) with $1 \leq i < j \leq n$.

```
node calendar(clk_1, …, clk_n : bool) returns (ok : bool);
  ok = qs_dfa(clk_1, clk_2) and … and qs_dfa(clk_n-1, clk_n);
tel
```

The node *N* of the topmost component type or component implementation has all clocks used by subcomponents as inputs and we wrap this node in another node *main* that has the same signature as *N* and add a stream and two assertions

```
clocks_are_quasi_synchronous = calendar(clk₁, …, clkₙ);
assert clocks_are_quasi_synchronous;
assert (clk1 or … or clkn);
```

This constrains the input clocks to be quasi-synchronous and ensures progress by forcing at least one clock to tick at each step.

A.8 Specification of Properties

Properties to be proved are specified in a file with the ending *.lustre_props* that is a Lustre file containing nodes as well as one special node that contains the properties to prove. This node is delimited by the keywords *node_properties* and *tel*, it does not have inputs or returns. Local variables can be declared and as usual, in addition a statement *check* that is similar to the *assert* statement marks a stream as a property to be proved.

The translator reads a *.lustre_props* file and adds each plain node from this file to the translated Lustre file in verbatim. Each stream defined in the *node_properties* section is added to the *main* node and each *check* statement is turned into a --*%PROPERTY* comment for the model checker.

A.9 Using the Translator

The translator is available at http://github.com/kind-mc/AADL2Kind, where installation instructions are provided.

The menu entry of the translator plug-in is only activated when a component *implementation* is selected in the outline view in the Eclipse IDE. To open the outline view, use *Window* > *Show View* > *Outline* in the menu bar.

To translate an AADL component, right click on the component implementation and select AADL to Kind from the pull-down menu. A Lustre file will be generated in a *lus* directory in the project.

A.10 Code Overview

The menu entry of the translator plug-in is only activated when a component implementation is selected in the outline view of eclipse IDE. Then the selected component implementation is fed to the *runjob* method in class *Aadl2kindAction* to start the translation process.

A.10.1 Data structures

CTNode is used to store the intermediate representation of a Lustre node translated from a component type.

CINode is used to store the intermediate representation of a Lustre node translated from a component implementation. Since a component implementation is an implementation of a component type, *CINode* is a subclass of *CTNode*, where its input ports, output ports Behavior Annex are declared.

BAnnex, State and Transition along with Equation are used to store an intermediate representation of Lustre stream definitions.

PropertySetEntity is used to store an intermediate representation of AADL properties defined in property set.

BoolType, *IntType*, *RealType*, *StringType* and *RecordType* correspond to the respective Lustre types, where the types record, string and real are not fully supported yet.

PortOccurence is used to create a port occurrence for a port used in a connection for later reference. *PortOccurence* along with *IdExpr* are used to store the port information, including port name, type, which component type it belongs to, and under what context it occurs.

A.10.2 Translation Process

With a component implementation passed to the *runjob* method in the *Aadl2kindAction* class, *ParseLibNodes.parseLib* is first called to parse library node files including quasi-synchronous clocks and other predefined Lustre nodes. The Lustre property file ending with *.lustre_props* is processed using the XText library that is also used by OSATE. The static fields of *Utils* are populated after that. Then the actual translation process starts by invoking the constructor of *Translator*.

In the *Translator* class, the method *traverseAADLModel* is used to traverse the AADL model in a bottom-up manner. Whenever a component type or component implementation is encountered, the methods *createCTNode* and *createCINode* are called, respectively, to instantiate an object of class *CTNode* or *CINode*. If there is an optimized predefined Lustre node for the component, it is used and put in the hashtable *Utils.staticNodes* instead of the generic translation.

In the method *createCTNode*, ports and properties are parsed by calling *parsePorts* and *parseCTPropertyAssociation* respectively. Since we are assuming that every component has a clock associated with it, a Boolean input for this clock is added in this parsing process. The static field *CICTPropertySet* in class *ParseAadl* is used to store a mapping from a component name qualified with the name of its package name to a list of properties used in that component. The parsing process of ports and properties for a component implementation is done in a similar fashion in the method *createCINode*.

If a component has a behavior specification, *parseBAnnex* in *ParseAadl.java* is called in *createCTNode* or *createCINode* to process the behavior specification in that component. Then an object of *BAnnex* is instantiated, with states, transition names and all local variables of that behavior specification as well as data subcomponents of the component implementation as local variables of a corresponding Lustre node. *ParseBehaviorActionBlock*, *parseBasicAction*, *parseAssignmentAction*, *ParseBehaviorActionSequenceorSet* along with some other auxiliary methods are used to process transitions in behavior specification. In this process, all the transition guards and assignment actions are eventually flattened to strings.

In the method *createCINode*, properties associated with subcomponents such as clock names and initial values for data subcomponents are parsed and propagated by calling the method *injectPropertyIntoSubcomponent*. Those properties are translated as inputs of a Lustre node if they are used in a component.

An additional step is needed in *createCINode* to parse the port connections in the component implementation. This is done by *parsePortConnections* in the class *ParseAadl*, where subcomponent port connections are translated into node calls and the output ports are translated to output streams without ports of other subcomponent as its definition in the Lustre node.

After all the components have been translated to Lustre nodes, *Utils.printNodes* is called to print all nodes to a buffered string, which is then written to a Lustre file with the same name as the topmost component type in the *lus* directory.